ELSEVIER

Theoretical Computer Science 206 (1998) 1–50

Theoretical Computer Science

Fundamental Study

Unification with extended patterns

Dominic Duggan*

Department of Computer and Engineering Science, Case Western Reserve University, 10900 Euclid Avenue, Cleveland, OH 44106, USA

Received March 1994; revised February 1995
Communicated by U. Montanari

Abstract

An extension of Miller's algorithm for β_0-unification of typed λ-terms is presented. The extension considered is the addition of products and polymorphism to β_0-unification; the new unification problem is termed $\beta_0\pi$-unification. Miller's pattern restriction is generalized by allowing repeated occurrences of variables to appear as arguments to free function variables, provided such variables are prefixed by distinct sequences of projections. This extended pattern restriction has several applications, including the definition of higher-order explicit substitutions. The algorithm is verified to terminate, and is shown to be sound and complete.

Contents

* E-mail: dduggan@acm.org.

0304-3975/98/$19.00
PII S0304-3975(97)00141-2

1. Introduction

Higher-order unification involves solving equality constraints between terms in a λ-calculus. Such forms of equality constraints are useful in applications involving the manipulation of syntax with variable-binding constructs, for example in predicate logic theorem provers and program transformers. Higher-order unification is the computational basis for the λ-Prolog and Elf logic programming languages [18, 17, 24], and forms a crucial component of the Isabelle generic theorem prover [22]. The complication with this form of unification is the presence of "free" function variables in applications, requiring any unification algorithm to synthesize λ-terms which may make the equated terms β-reduce to equal terms. Unlike first-order unification, solving such constraints is in general undecidable. Even if the equated terms are unifiable, there may not be a single most general unifier; in fact there may be an infinite number of incomparable unifiers. Huet [12] provided the most widely known algorithm for higher-order preunification,[1] while Elliot [5, 6] and Pym [27] have extended Huet's algorithm to a λ-calculus with dependent types but without polymorphism.

Miller [14] has recently discovered a restricted subset of higher-order unification which enjoys many of the same properties as first-order unification. This restricted subset is referred to as β_0-*unification*. β_0-unification is guaranteed to terminate, and if the equated terms are unifiable then there is a most general unifier[2] which unifies them. The heart of β_0-unification is to limit the forms of applications of "free" function variables such that β-reduction amounts to permutation of "local" (λ-bound) variables in a term. These restricted applications are called *patterns*, since they specify the λ-bound variables which may occur in the result of instantiating a variable and β-reducing the result. For example, in the equality constraint:

$$\lambda x \cdot \lambda y \cdot Fx = \lambda x \cdot \lambda y \cdot \mathsf{c}\, x\, y$$

between two λ-terms, where c is a constant, the fact that F is applied to x but not to y means that x may occur in the result of instantiating F and β-reducing, but not y. Since the term it is equated to contains both x and y in the body, the terms are not unifiable. This restricted form of higher-order unification (or, extended form of first-order unification) is particularly suitable as a basis for manipulating "higher-order

[1] *Preunification* refers to the solution of a collection of equality constraints between trees without finally solving the constraints which only involve "free" variables at the head. One of the key insights in Huet's algorithm was to return these equality constraints unsolved without trying to enumerate all of the their unifying substitutions, content that the constraints are at least satisfiable.

[2] Most general unifiers are typically not unique, since free variables may be arbitrarily renamed (as in first-order unification) and also arguments may be ordered arbitrarily when unifying two terms with free variables at the heads (which does not occur in first-order unification).

abstract syntax," wherein variable binding in an object language is represented by λ-binding in a meta-language representation. In some sense it is the "interesting" subset of higher-order unification. Pfenning has extended β_0-unification to dependent type systems (including the Coquand–Huet Calculus of Constructions) and has also given an algorithm for anti-unification in dependent type systems [24]. Nipkow has used the algorithm to generalize many results from the theory of first-order rewrite systems to higher-order rewrite systems (for example providing a higher-order critical pairs lemma [20]).

Miller has also shown how higher-order unification may be coded as a logic program in a language which supports β_0-unification [15]. This application demonstrates both the power and the weakness of the pattern restriction. The power of unification restricted to patterns is that it separates the variable-binding aspects of higher-order unification from the search aspects; in many applications only the former is required. The weakness of this form of unification is that substitutions must be explicitly implemented in the metalanguage. Since the representation does not support the composition of substitutions, such an implementation leads to a heavy amount of copying and a corresponding loss in efficiency. Thus the Elf language, although it employs the pattern restriction for unification problems, implements substitution in the runtime, treating unification problems violating the restriction as "hard constraints" to be delayed until substitution has simplified the problem to a solvable form.

In this paper we present a further generalization of β_0-unification, to product types and polymorphism. The generalization to polymorphism is already contained in Pfenning's extension of the original algorithm to the Calculus of Constructions [25]. The real interest in our work is in generalizing Miller's algorithm to the combination of polymorphism and product types. One motivation is to improve the data structuring facilities of the metalanguage in which patterns are embedded. Combining unification with products and polymorphism has some interest independently of the main motivation for the present work, and this combination causes some serious technical problems which we overcome (as explained later in this section).

The central motivation for this work is that the addition of product types leads to a generalization of Miller's pattern restriction where λ-bound variables may be repeated in the application of a function variable. The crucial restriction is that each such variable have a distinct prefix of projectors applied to it. We refer to this as the *extended pattern restriction*, and we refer to the unification problem as *$\beta_0\pi$-unification*. For example, with the equality constraint:

$$\lambda x \cdot F(\mathsf{fst}\ x)(\mathsf{snd}(\mathsf{snd}\ x)) = \lambda x \cdot \mathsf{c}(\mathsf{snd}(\mathsf{fst}\ x))(\mathsf{snd}\ x)$$

the occurrence of $\mathsf{snd}(\mathsf{fst}\, x)$ in the right-hand side is allowed because $\mathsf{fst}\ x$ is an argument to the function variable F. On the other hand $\mathsf{snd}\, x$ in the right-hand side violates the scoping restrictions of the constraint, since neither x nor $\mathsf{snd}x$ are arguments to F.

$\beta_0\pi$-unification provides several interesting generalizations of the capabilities of β_0-unification. One of its principal applications is in the definition of a higher-order form

of the $\lambda\sigma$-calculus of explicit substitutions [2]. The $\lambda\sigma$-calculus is based on a first-order representation of abstract syntax, manipulating object language variables as de-Bruijn numbers [7, 1, 19]. The difficulty with defining a version of the $\lambda\sigma$-calculus using higher-order abstract syntax, is in how to compose the substitutions when collapsing two nested closures into a single closure. Extended patterns provide the critical ingredient that allows this technical obstacle to be overcome. Further details for this higher-order version of the $\lambda\sigma$-calculus are provided by Duggan [2].

The definition of higher-order explicit substitutions relies critically on both products and polymorphism in the metalanguage. A similar metalanguage for higher-order abstract syntax was originally proposed by Pfenning and Elliott [26], motivated by the need to represent object language variable-binding constructs of arbitrary arity (e.g., multiary functions). Pfenning and Elliott's metalanguage combined products, polymorphism and higher-order abstract syntax, but they did not provide an algorithm to support their examples. The algorithm for $\beta_0\pi$-unification presented here may be used to program their examples, as explained in [3].

Elliott [6] considered the addition of (dependent) product types to his algorithm for unification with (first-order) dependent function types. In this case (without polymorphism) it is possible to η-expand all atomic terms of product type, replace all variables of product type with pairs of new variables (thus removing all "flexible" occurrences of projectors $\mathsf{fst}\ M$ and $\mathsf{snd}\ M$), and use the following pair–pair rule to simplify away the resulting pairs:

$$(M_1, M_2) = (N_1, N_2) \quad \text{becomes } M_1 = N_1,\ M_2 = N_2.$$

The addition of polymorphism introduces non-trivial complications; since type variables may be instantiated during unification by product types, it is not possible to remove occurrences of projectors by preprocessing the initial constraints. Whereas Huet's algorithm only considers atomic normal forms of the form $(x\,M_1 \ldots M_n)$, it is necessary to provide an alternative but equally uniform normal form in the presence of projectors. In Section 2 we introduce a "locator calculus" that provides this normal form. There are subtle complications in reasoning about termination and considering where to perform the occurs check. The complication with projector terms is that in certain circumstances the occurrence of F in the term it is equated to does not necessarily entail non-unifiability. Consider the following constraints:

$$F = (\mathsf{c}_1, \mathsf{c}_2(\mathsf{fst}\,F)) \tag{1}$$

$$F = (\mathsf{fst}\,F, \mathsf{fst}\,F) \tag{2}$$

$$F = (\mathsf{fst}\ (\mathsf{fst}\,F), \mathsf{fst}\,F) \tag{3}$$

$$F = (\mathsf{snd}\,F,\ \mathsf{c}(\mathsf{fst}\,F)). \tag{4}$$

In the first equation above, the occurrence of $\mathsf{fst}\,F$ in the second component of the pair refers to the left component of the pair. Thus if we consider terms as finite trees, compositions of projectors give a form of path-addressing, effectively turning trees into

directed graphs. Provided no cycles arise in such addressing, unification should not fail due to such self-referential occurrences of the free variables. The second equation similarly exemplifies the case where occurrences of the variable in the term to which it is equated do not entail non-unifiability. Here the first occurrence of $\mathsf{fst}\,F$ simply refers to the first component of the pair itself. In Section 5 we refer to such occurrences as "non-recursive self-references".

The third equation exemplifies a "recursive self-reference": in this case, the first component of the pair is equated to its own left subtree, and this circularity makes the terms non-unifiable. We refer to this as a "non-rigid recursive self-reference", since the circularity does not involve the use of constants or λ-bound variables at the head of an application containing a recursive self-reference. Such recursive self-references are disallowed by the type system; if they were not, it would be necessary to include an occurs check in the case where a variable is equated with a pair, and the first two equations demonstrate that this would be more complicated than the usual occurs check in first-order unification and in Miller's algorithm.

Finally the fourth equation exemplifies a "rigid recursive self-reference": in this case the second component of the pair is a rigid application containing a subterm equated to the left component, however the first component is in turn equated to the second component. This example illustrates a recursive self-reference which (a) leads to a circularity which makes the terms non-unifiable, (b) is not disallowed by the type system (provided c has the appropriate type), and (c) requires some unification to be performed before it becomes evident. In this case the circularity will be caught by the usual occurs check when unifying a new variable (introduced to represent the second component) with the rigid application with head c. The occurs check which detects this circularity may not be performed until after pair-expansions are performed, replacing free variables by pairs of free variables. An example of this is given at the beginning of Section 5. Note that rigid recursive self-references do not require an occurs check when unifying a variable with a pair, significantly simplifying the algorithm. Section 7 contains some further discussion of this point.

In the next section we present the type theory which we use as the basis for our algorithm. The type system is Girard's System $\mathbf{F}^{\omega}$ with simple product types. Alternative type systems other than System $\mathbf{F}^{\omega}$ could have been chosen. One such candidate is the Coquand–Huet calculus of constructions considered by Pfenning. The advantage of the latter calculus is that terms and types have a common abstract syntax, leading to a more economical notation than the calculus used here. On the other hand, dependent product (general sum) types raise some subtle complications in a system with impredicative polymorphism, while being of dubious practical value. Furthermore, the meta-theory of constructions is considerably more complicated, because of the dependence of types on terms. For example, confluence under $\beta\eta$-conversion in constructions was only recently verified [9], while it is a straight-forward application of Girard's "Candidates sur le Reducibilité" for System $\mathbf{F}^{\omega}$ [8]. For these reasons we have chosen to use a system of simple and polymorphic types, without dependent types. Finally, the decision to use impredicative polymorphism, rather than say a two-level system of

predicative polymorphism, was taken to simplify the presentation (we do not need to differentiate between "big" and "small" types, or alternatively to introduce a lifting of types to kinds which would over-complicate the equality theory [11]).

Products are definable in the impredicative calculus:

$$A \times B \stackrel{\text{def}}{=} \Lambda t : \mathsf{Type} \cdot (A \to B \to t) \to t$$

$$(M,N) \stackrel{\text{def}}{=} \Lambda t : \mathsf{Type} \cdot \lambda f : A \to B \to t \cdot f\, M\, N$$

$$\mathsf{fst}\, M \stackrel{\text{def}}{=} M\,[A]\,(\lambda x : A \cdot \lambda y : B \cdot x)$$

$$\mathsf{snd}\, M \stackrel{\text{def}}{=} M\,[B]\,(\lambda x : A \cdot \lambda y : B \cdot y).$$

There are several reasons why this is insufficient for our purposes. First, terms of the form $w[A](\lambda x : A \cdot \lambda y : B \cdot x)$ and $w[B](\lambda x : A \cdot \lambda y : B \cdot y)$ violate Miller's restrictions on what may occur as arguments to a free function variable; it is not clear how to formulate a generalization of Miller's restriction to allow such λ-terms as arguments. Second, surjective pairing does not hold for this calculus, which is basically essential for a usable algorithm; for example it justifies the use of the **Pair-Atom, Loc-Elim** and **Pair-Elim** rules in Section 4. Finally, we would like our results to be applicable to other forms of polymorphism, such as the two-level predicative system mentioned earlier, and such encodings are not possible in such predicative systems.

Section 2 presents the underlying calculus that we consider; we refer to this as the "locator calculus." This calculus is System $\mathbf{F}^{\omega}$ extended with product types; products are presented in a slightly non-traditional manner, in order to provide a useful normal form for the calculus. Appendix A briefly explains the relationship between this calculus and the more usual presentation of products in λ-calculi. In Section 3 we define our extension of Miller's pattern restriction, and show that it has a similar notion of a restricted β-reduction. In Section 4 we present our algorithm, while we present its proof of correctness in Section 5 (termination) and Section 6 (soundness and completeness). Because of the complications introduced by the combination of products and polymorphism, reasoning about termination constitutes the main technical problem with verifying the algorithm. We develop a new technique for proving termination, using the size of terms with sharing. Section 7 provides our conclusions.

2. Product normal form and the locator calculus

2.1. Environments and types

In this section we present the type theory underlying our calculus. We use judgements of the form $\Gamma\,\mathbf{env}_{\Sigma}$ and $\Gamma \rhd_{\Sigma} \mathscr{F}$, where Σ is a signature of typings for constants, Γ is

$$\textsc{Const}\quad \frac{\mathsf{tc}\in dom(\Sigma)}{\Gamma \rhd_\Sigma \mathsf{tc}\in\Sigma(\mathsf{tc})} \qquad \frac{\mathsf{c}\in dom(\Sigma)}{\Gamma \rhd_\Sigma \mathsf{c}\in\Sigma(\mathsf{c})}$$

$$\textsc{Env}\quad \frac{}{\mathsf{nil}\ \mathbf{env}_\Sigma} \qquad \frac{\Gamma \rhd K\ \mathbf{kind}_\Sigma\ (t\notin dom(\Gamma))}{\Gamma, t:K\ \mathbf{env}_\Sigma} \qquad \frac{\Gamma \rhd_\Sigma A\in \mathsf{Type}\ (x\notin dom(\Gamma))}{\Gamma, x:A\ \mathbf{env}_\Sigma}$$

$$\textsc{Var}\quad \frac{\Gamma \mathbf{env}_\Sigma\ t\in dom(\Gamma)}{\Gamma \rhd_\Sigma t\in\Gamma(t)} \qquad \frac{\Gamma \mathbf{env}_\Sigma\ x\in dom(\Gamma)}{\Gamma \rhd_\Sigma x\in\Gamma(x)}$$

Fig. 1. Formation rules for environments.

$$\textsc{Type}, \to F\quad \frac{\Gamma \mathbf{env}_\Sigma}{\Gamma \rhd \mathsf{Type}\ \mathbf{kind}_\Sigma} \qquad \frac{\Gamma \rhd K_1\ \mathbf{kind}_\Sigma \Gamma \rhd K_2 \mathbf{kind}_\Sigma}{\Gamma \rhd K_1 \to K_2\ \mathsf{kind}_\Sigma}$$

$$\to I, \to E\quad \frac{\Gamma, t:K_1 \rhd_\Sigma A\in K_2}{\Gamma \rhd_\Sigma \lambda t:K_1\cdot A\in K_1\to K_2} \qquad \frac{\Gamma \rhd_\Sigma A\in K_1\to K_2 \quad \Gamma \rhd_\Sigma B\in K_1}{\Gamma \rhd_\Sigma (A\,B)\in K_2}$$

$$\to\beta, \to\eta\quad \frac{\Gamma, t:K_1 \rhd_\Sigma A\in K_2 \quad \Gamma \rhd_\Sigma B\in K_1}{\Gamma \rhd_\Sigma (\lambda t:K_1\cdot A)B=\{B/t\}A} \qquad \frac{\Gamma \rhd_\Sigma A\in K_1\to K_2 \quad t\notin dom(\Gamma)}{\Gamma \rhd_\Sigma \lambda t:K_1\cdot At=A}$$

Fig. 2. Formulation rules for types and type operators.

a typing environment and $\mathscr{F}$ a formula of the meta-logic. Formulae of the meta-logic and terms of the language are organized into the following categories:

$$\mathscr{F}\in \text{Formulae} ::= K\ \mathbf{kind}_\Sigma \mid A\in K \mid M\in A \mid A=B \mid M=N$$
$$\Gamma\in \text{Envs} ::= \mathsf{nil} \mid \Gamma, t:K \mid \Gamma, x:A$$
$$K\in \text{Kinds} ::= \mathsf{Type} \mid K_1\to K_2$$
$$A,B\in \text{Types} ::= \mathsf{tc} \mid t \mid A\times B \mid A\to B \mid \Delta t:K\cdot A \mid \lambda t:K\cdot A \mid (A\,B).$$

Type judgements are made relative to a signature Σ. Such a signature has at least the type constructor $\to$ and $\times$ of kind $\mathsf{Type}\times\mathsf{Type}\to\mathsf{Type}$, and the family of type constructors Δ of kind $(\kappa\to\mathsf{Type})\to\mathsf{Type}$ (implicitly indexed by the kind κ). This signature may contain other type and term constants; we use tc and c to denote type and term constants, respectively. Fig. 1 gives the formation rules for environments. We separate the namespace into term variables and type and type operator variables. Fig. 2 gives the formation rules for the types.[3] Essentially the types constitute a simply typed λ-calculus (our calculus is based on Girard's System $\mathbf{F}^\omega$). These formation rules are formulated independently of the rules for terms.

[3] To be more precise, these give formation rules for types and type operators. For most of the paper we will use "types" as a semantic abbreviation for "types and type operators".

$$\rightarrow F, \Delta F \qquad \frac{\Gamma \rhd_\Sigma A \in \mathsf{Type} \quad \Gamma \rhd_\Sigma B \in \mathsf{Type}}{\Gamma \rhd_\Sigma A \rightarrow B \in \mathsf{Type}} \qquad \frac{\Gamma \rhd_\Sigma A \in \mathsf{Type} \quad \Gamma \rhd_\Sigma B \in \mathsf{Type}}{\Gamma \rhd_\Sigma \Delta t : K \cdot A \in \mathsf{Type}}$$

$$\rightarrow I, \Delta I \qquad \frac{\Gamma, x : A \rhd_\Sigma M \in B}{\Gamma \rhd_\Sigma \lambda x : A \cdot M \in A \rightarrow B} \qquad \frac{\Gamma, t : K \rhd_\Sigma M \in A}{\Gamma \rhd_\Sigma \Lambda t : K \cdot M \in \Delta t : K \cdot A}$$

$$\rightarrow E, \Delta E \qquad \frac{\Gamma \rhd_\Sigma M \in A \rightarrow B \quad \Gamma \rhd_\Sigma N \in A}{\Gamma \rhd_\Sigma (M\,N) \in B} \qquad \frac{\Gamma \rhd_\Sigma M \in \Delta t : K \cdot A \quad \Gamma \rhd_\Sigma B \in K}{\Gamma \rhd_\Sigma M[B] \in \{B/t\}A}$$

$$\rightarrow \beta, \Delta \beta \qquad \frac{\Gamma, x : A \rhd_\Sigma M \in B \quad \Gamma \rhd_\Sigma N \in A}{\Gamma \rhd_\Sigma (\lambda x : A \cdot M)\,N = \{N/x\}M} \qquad \frac{\Gamma, t : K \rhd_\Sigma M \in A \quad \Gamma \rhd_\Sigma B \in K}{\Gamma \rhd_\Sigma (\Lambda t : K \cdot M)\,[B] = \{B/t\}M}$$

$$\rightarrow \eta, \Delta \eta \qquad \frac{\Gamma \rhd_\Sigma M \in A \rightarrow B \quad x \notin dom(\Gamma)}{\Gamma \rhd_\Sigma \lambda x : A \cdot Mx = M} \qquad \frac{\Gamma \rhd_\Sigma M \in \Delta t : K \cdot A \quad t \notin dom(\Gamma)}{\Gamma \rhd_\Sigma \Lambda t : K \cdot M[t] = M}$$

$$\textsc{TyConv} \qquad \frac{\Gamma \rhd_\Sigma M \in A \quad \Gamma \rhd_\Sigma A' \in \mathsf{Type} \quad \Gamma \rhd_\Sigma A = A'}{\Gamma \rhd_\Sigma M \in A'}$$

Fig. 3. Type rules for functional term calculus.

We are only concerned with terms and types in $\beta\eta$-normal form and product normal form ($\beta\eta\delta$-normal form, defined below). We will not consider terms in long $\beta\eta$-normal form (i.e. with terms η-expanded so that every atom of function type is the function part of an application, while every atom of product type is the target of a projector). In Section 3, we find it easier and simpler to specify the restrictions on terms in $\beta\eta$-normal form rather than long normal form. Furthermore, requiring terms to be preprocessed to be in long normal form is insufficient for our algorithm, since function and product types may be introduced (by the instantiation of "free" type variables) during the execution of the algorithm.

2.2. *The functional term calculus*

We now consider the term calculus for our type theory. In this section we only consider the subset of the calculus with functional and polymorphic types ($\rightarrow$ and Δ, respectively). In the next section we extend this with products, with a moderately non-traditional presentation. The functional subset of this calculus is Girard's system $\mathbf{F}^\omega$, a polymorphic λ-calculus with abstraction over types and type operators:

$$M, N ::= \mathsf{c} \mid x \mid \lambda x : A \cdot M \mid (M\,N) \mid \Lambda t : K \cdot M \mid M\,[A].$$

The type rules for this term calculus, which are completely standard, are provided in Fig. 3. The reduction relation for this calculus includes the usual β and η-reduction rules.

We use the following notational conventions: we use $\overline{M_n}$ to denote the sequence of terms $M_1, \ldots, M_n$ (similarly for other syntactic classes), while we use $[n]$ to denote the

set $\{1,\ldots,n\}$. We use $\Lambda\overline{t_m}:\overline{K_m}\cdot M$ to denote $\Lambda t_1:K_1\cdots\Lambda t_m:K_m\cdot M$, $\lambda\overline{x_m}:\overline{A_m}\cdot M$ to denote $\lambda x_1:A_1\cdots\lambda x_m:A_m\cdot M$, $\lambda\overline{t_m}:\overline{K_m}\cdot A$ to denote $\lambda t_1:K_1\cdots\lambda t_m:K_m\cdot A$, $M[\overline{A_m}]$ to denote $M[A_1]\ldots[A_m]$, $(M\overline{N_m})$ to denote $(MN_1\ldots N_m)$, and $(A\overline{B_m})$ to denote $(AB_1\ldots B_m)$. Furthermore, we use $\overline{K_m}\to K$ to denote $K_1\to\cdots\to K_m\to K$ and $\overline{A_m}\to B$ to denote $A_1\to\cdots\to A_m\to B$. We denote term variables by f,g,h,w,x,y,z,F,G,H and type variables by t,u,v,T,U,V.

The β and η rules give rise to an equality theory on the underlying term calculus. We omit the statement of the usual congruence and closure rules which complete the statement of this theory, as well as the reflexive, symmetric and transitive rules. Omitting the rules for symmetry gives rise to a rewriting system; let $\Gamma\vdash_\Sigma M\twoheadrightarrow N\in A$ denote that $\Gamma\rhd_\Sigma M\in A$ and N results from applying these rewritting rules to M, similarly for types $\Gamma\vdash_\Sigma\twoheadrightarrow B\in K$. The following theorem summarizes the meta-properties we require of this calculus:

Theorem 1. *The following properties hold for the calculus considered here:*

Confluence for types: *If* $\Gamma\vdash_\Sigma B_1\in K$, $\Gamma\vdash_\Sigma B_2\in K$, $\Gamma\vdash_\Sigma A\twoheadrightarrow B_1\in K$ *and* $\Gamma\vdash_\Sigma A\twoheadrightarrow B_2\in K$, *then there exists some* A' *such that* $\Gamma\vdash_\Sigma B_1\twoheadrightarrow A'\in K$ *and* $\Gamma\vdash_\Sigma B_2\twoheadrightarrow A'\in K$.

Subject reduction for types: *If* $\Gamma\vdash_\Sigma A\twoheadrightarrow A'\in K$, *then* $\Gamma\vdash_\Sigma A'\in K$.

Strong normalization for types: *There does not exist an infinite sequence of rewrite steps* $\Gamma\vdash_\Sigma A_i\twoheadrightarrow A_{i+1}\in K$ *for* $i\in\omega$.

Confluence for terms: *If* $\Gamma\vdash_\Sigma N_1\in A$, $\Gamma\vdash_\Sigma N_2\in A$, $\Gamma\vdash_\Sigma M\twoheadrightarrow N_1\in A$ *and* $\Gamma\vdash_\Sigma M\twoheadrightarrow N_2\in A$, *then there exists some* M' *such that* $\Gamma\vdash_\Sigma N_1\twoheadrightarrow M'\in A$ *and* $\Gamma\vdash_\Sigma N_2\twoheadrightarrow M'\in A$.

Subject reduction for terms: *If* $\Gamma\vdash_\Sigma M\in A$ *and* $\Gamma\vdash_\Sigma M\twoheadrightarrow M'\in A$, *then* $\Gamma\vdash_\Sigma M'\in A$.

Strong normalization for terms: *There does not exist an infinite sequence of rewrite steps* $\Gamma\vdash_\Sigma M_i\twoheadrightarrow M_{i+1}\in A$ *for* $i\in\omega$.

Uniqueness of types: *If* $\Gamma\vdash_\Sigma M\in A$ *and* $\Gamma\vdash_\Sigma M\in B$, *then* $\Gamma\vdash_\Sigma A=B$.

Decidability of type-checking: *The validity of the judgements* $\Gamma\rhd_\Sigma A\in K$ *and* $\Gamma\rhd_\Sigma M\in A$ *is decidable. Given a type* A, *denote its unique kind by* $\tau_\Gamma(A)$. *Given a term* M, *denote the unique normal-form representative for the equivalence class of its type by* $\tau_\Gamma(M)$.

The results for types are standard for the simply-typed λ-calculus. The results for terms (in particular confluence and strong normalization) were verified by Girard in his Ph.D. thesis [10] (see also [8]).

The terms in $\beta\eta$-normal form in this functional subset of the term calculus have the form:

$$P::=\mathsf{c}\mid x\mid(P_1\,P_2)\mid P[A],$$

$$Q::=P\mid\lambda x:A\cdot Q\mid\Lambda t:K\cdot Q,$$

where P is the class of *atomic* normal form terms. A notational complication here is how to denote an arbitrary term in normal form (for the simply typed subset of this

calculus, $\lambda\overline{x_m}:\overline{A_m}\cdot M\overline{N_n}$ would suffice). We will use $\Lambda\overline{t_{m_1}}:\overline{K_{m_1}}\cdot\lambda\overline{x_{m_2}}:\overline{A_{m_2}}\cdot y[\overline{B_{n_1}}]\,\overline{M_{n_2}}$ to denote a term in normal form with an initial prefix of λ and Λ abstractions binding $t_1,\ldots,t_{m_1},x_1,\ldots,x_{m_2}$, and with head variable y applied to $B_1,\ldots,B_{n_1},M_1,\ldots,M_{n_2}$. No confusion in variable binding arises because of this (notational) permutation of types and terms, since types do not depend on term variables. We emphasize that this permutation is purely notational (Harper et al give a modules calculus with an underlying equality theory which makes this form of rearrangement explicit in the calculus [11]). We will denote the type of such a term by $\Lambda\overline{t_{m_1}}:\overline{K_{m_1}}\cdot\overline{A_{m_2}}\to B$. Extending our earlier notation, we use $\overline{K_m}$, $\overline{A_n}$ to denote a sequence of interleavings of kinds and types, and $\overline{A_m}$, $\overline{M_n}$ to denote a sequence of types and terms.

2.3. The locator term calculus

We now consider the notational complications introduced by products, and give a non-traditional presentation of System $\mathbf{F}^\omega$ extended with products which solves these problems. Appendix A considers a traditional extension of the type system for System $\mathbf{F}^\omega$ with products. The syntax for terms is extended with:

$$M,N ::= \ldots \mid (M,N) \mid \mathsf{fst}\,M \mid \mathsf{snd}\,M$$

where we refer to $\mathsf{fst}\,M$ and $\mathsf{snd}\,M$ as *projectors* (and we refer to the traditional presentation of System $\mathbf{F}^\omega$ with products as the *projector calculus*). This has the usual β and η-reduction rules:

$$\mathsf{fst}\,(M,N)\twoheadrightarrow_\beta M,\qquad \mathsf{snd}\,(M,N)\twoheadrightarrow_\beta N,\qquad (\mathsf{fst}\,M,\mathsf{snd}\,M)\twoheadrightarrow_\eta M$$

The terms in normal form have the form:

$$P ::= \mathsf{c} \mid x \mid (P_1\,P_2) \mid P[A] \mid \mathsf{fst}\,P \mid \mathsf{snd}\,P$$

$$Q ::= P \mid \lambda x:A\cdot Q \mid \Lambda t:K\cdot Q \mid (Q_1,Q_2)$$

Although this might well be the usual familiar calculus in the interface for tools based on our algorithm, we do not use this calculus as the basis for our unification algorithm, because of the complicated atomic normal forms. We will use an idea originally proposed by Elliott [4], placing terms (and types and type operators) in *product normal form*. In the interests of defining this normal form, we introduce new constants to replace projectors in the term calculus. We introduce conversion rules involving these constants, whose admissibility is verified by considering their translation into combinators in the original calculus. We refer to this new calculus as the *locator calculus*.

Definition 2 (*Selectors and locators*). The original term calculus is extended as follows. We refer to the numerals **1** and **2** as selectors. A locator L is a (possibly empty) sequence of selectors. We denote the empty or identity locator by **id**, and we denote concatenation of locators by $L_1\ L_2$ (juxtaposition). $L_1\leqslant L_2$ denotes that L_1 is a prefix of L_2.

The *locator calculus* is defined by using locators rather than projectors (as in Appendix A). So rather than terms of the form $\mathsf{fst}\,M$ and $\mathsf{snd}\,M$, we deal with terms of the form $L\,M$. Locators have a stronger equality theory than projectors, reflecting the fact that the corresponding reduction relation is intended as an algorithm for converting terms to *product normal form*.

In Appendix A we verify that the locator calculus is a conservative extension of System $\mathbf{F}^{\omega}$ with products, in the sense that there is a equality-preserving mapping from terms in the locator calculus to terms in the projector calculus, such that well-typed terms in the former are mapped to well-typed terms in the latter. The basic idea is to extend the reduction relation to allow applications of locators to be pushed to the "head" of an application, so $L\,(M\,N) \twoheadrightarrow (L\,M)\,N$ and $L\,(M\,[A]) \twoheadrightarrow (L\,M)\,[A]$. However polymorphism introduces some tedious complications. Assume for example that the signature contains a constant c of type $\Delta t : \mathsf{Type} \cdot t$, then we would like the following conversion to hold:

$$\mathbf{1}\,(\mathsf{c}\,[A \times B]) = (\mathbf{1}\,\mathsf{c})\,[A \times B].$$

To validate this conversion, we introduce (in Appendix A) a translation for terms $[\![\Gamma \vdash_{\Sigma} M \in A]\!]$ with

$$[\![\Gamma \vdash_{\Sigma} \mathbf{1}\,(\mathsf{c}\,[A \times B]) \in A]\!] = (\lambda f : A \times B \cdot \mathsf{fst}\,f)\ (\mathsf{c}\,[A \times B])$$

$$[\![\Gamma \vdash_{\Sigma} (\mathbf{1}\,\mathsf{c})\,[A \times B] \in A]\!]$$

$$= (\lambda f : (\Delta t : \mathsf{Type} \cdot t) \cdot \Lambda t : \mathsf{Type} \cdot \mathsf{fst}(f\,[A \times B]))\,\mathsf{c}\,[A \times B].$$

The main point to note here is that the translation (and the typing of located atoms) must be made relative to a surrounding context of polymorphic applications.

Definition 3 (*Locator calculus*). The *locator calculus* is given by extending the functional term calculus given in the previous subsection with the following rules:

$$L \in \mathit{Locators} ::= \mathbf{id} \mid \mathbf{1} \mid \mathbf{2} \mid L_1\,L_2$$

$$M, N \in \mathit{Terms} ::= \ldots \mid (M, N) \mid L\,M$$

The type and equality rules for this term calculus are given by extending the rules in Figs. 1–3 with the rules in Fig. 4. For convenience we identify terms M and located terms $\mathbf{id}\,M$. The last δ-rule gives us the associativity $L\,(L'\,M) = (L\,L')\,M$.

Appendix A defines the translation $[\![\Gamma \vdash_{\Sigma} M \in A]\!]$ (by induction over type derivations) from the locator calculus into the projector calculus i.e., System $\mathbf{F}^{\omega}$ with products. The equivalence between the two systems is given by

Theorem 4. (i) *$\Gamma \vdash_{\Sigma} M \in A$ if and only if $\Gamma \vdash_{\Sigma} [\![\Gamma \vdash_{\Sigma} M \in A]\!] \in A$.*
(ii) *$\Gamma \vdash_{\Sigma} M = N$ if and only if $\Gamma \vdash_{\Sigma} [\![\Gamma \vdash_{\Sigma} M \in A]\!] = [\![\Gamma \vdash_{\Sigma} N \in A]\!]$.*

$$\times F, \times I \quad \frac{\Gamma \rhd_\Sigma A \in \mathsf{Type} \quad \Gamma \rhd_\Sigma B \in \mathsf{Type}}{\Gamma \rhd_\Sigma A \times B \in \mathsf{Type}} \quad \frac{\Gamma \rhd_\Sigma M \in A \quad \Gamma \rhd_\Sigma N \in B}{\Gamma \rhd_\Sigma (M, N) \in A \times B}$$

$$\times E \quad \frac{\Gamma \rhd_\Sigma M \in A}{\Gamma \rhd_\Sigma (\mathbf{id}\, M) \in A} \quad \frac{\Gamma \rhd_\Sigma (L\, M[\overline{B_p}]\overline{N_q}) \in \Delta \overline{t_k} : \overline{K_k} \cdot \overline{A_m} \to A_1' \times A_2'}{\Gamma \rhd_\Sigma ((i\, L)\, M[\overline{B_p}]\overline{N_q}) \in \Delta \overline{t_k} : \overline{K_k} \cdot \overline{A_m} \to A_i'}$$

$$\times \beta \quad \frac{\Gamma \rhd_\Sigma L\, M_1 \in A_1 \quad \Gamma \rhd_\Sigma M_2 \in A_2}{\Gamma \rhd_\Sigma (L\, \mathbf{1})(M_1, M_2) = (L\, M_1)} \quad \frac{\Gamma \rhd_\Sigma M_1 \in A_1 \quad \Gamma \rhd_\Sigma L\, M_2 \in A_2}{\Gamma \rhd_\Sigma (L\, \mathbf{2})(M_1, M_2) = (L\, M_2)}$$

$$\times \eta \quad \frac{\Gamma \rhd_\Sigma M \in A}{\Gamma \rhd_\Sigma (\mathbf{id}\, M) = M}$$

$$\times \eta \quad \frac{\Gamma \rhd_\Sigma L\, M[\overline{A_m}]\overline{N_n} \in B_1 \times B_2}{\Gamma \rhd_\Sigma ((\mathbf{1}L)\, M[\overline{A_m}]\overline{N_n},\ (\mathbf{2}L)\, M[\overline{A_m}]\overline{N_n}) = L M[\overline{A_m}]\overline{N_n}}$$

$$\times \delta \quad \frac{\Gamma \rhd_\Sigma L\, (\Lambda t : K \cdot M) \in B}{\Gamma \rhd_\Sigma L\, (\Lambda t : K \cdot M) = \Lambda t : K \cdot L M} \quad \frac{\Gamma \rhd_\Sigma L\, (\lambda x : A \cdot M) \in B}{\Gamma \rhd_\Sigma L (\lambda x : A \cdot M) = \lambda x : A \cdot L M}$$

$$\times \delta \quad \frac{\Gamma \rhd_\Sigma L\, (M\, [A]) \in B}{\Gamma \rhd_\Sigma L\, (M\, [A]) = (L M)\, [A]} \quad \frac{\Gamma \rhd_\Sigma L\, (M\, N) \in A}{\Gamma \rhd_\Sigma L\, (M\, N) = L\, M\, N}$$

$$\times \delta \quad \frac{\Gamma \rhd_\Sigma L\, (L'\, M) \in A}{\Gamma \rhd_\Sigma L\, (L'\, M) = (L\, L')\, M}$$

Fig. 4. Type and equality rules for the locator calculus.

Definition 5 (*Product normal form*). The types and terms in *product normal form* (i.e. $\beta\eta\delta$-normal form) are described by the following grammar:

$$A ::= \Delta t : K \cdot A \mid A \to A' \mid A \times A' \mid t\overline{A} \mid \lambda t : K \cdot A$$

$$M ::= (L c\, [\overline{A_m}]\overline{M_n}) \mid (L x[\overline{A_m}]\overline{M_n}) \mid \lambda x : A \cdot M \mid \Lambda t : K \cdot M \mid (M_1, M_2)$$

We will denote (unlocated) type variables by p^t and q^t, and located term variables by p^x and q^x, in the remainder. We will use p and q to denote such terms where the syntactic class is unimportant.

2.4. Properties of the locator calculus

We need to verify that $\beta\eta\delta$-reduction is confluent and terminating. Let $\twoheadrightarrow_\delta$ denote reduction under the δ-rules.

Lemma 6. *$\twoheadrightarrow_\delta$ is canonical i.e., confluent and terminating.*

Termination is verified by using the multiset measure {size of $N \mid LN$ is a subterm of M}, which is reduced by each application of a δ-rule to M. Local confluence follows from the absence of critical pairs. Confluence is a result of local confluence and termination. Since normal forms are unique and always defined, we denote the normal form of M under δ-reduction by $\delta\mathtt{norm}(M)$. Now define the rewrite relation $M \rightrightarrows N$ by: $\delta\mathtt{norm}(M) \overset{*}{\rightrightarrows} \delta\mathtt{norm}(N)$, where $\twoheadrightarrow$ denotes $\beta\eta\delta$-reduction, and $\overset{*}{\twoheadrightarrow}$ denotes the reflexive transitive closure of the $\twoheadrightarrow$ relation (similarly for any other rewrite relation, such as $\rightrightarrows$).

Lemma 7. *If $M \twoheadrightarrow N$ then $M \overset{*}{\rightrightarrows} N$.*

Proof. The verification is by induction on the number of δ-redices in M, verifying that it is possible to permute δ-reductions with β-reductions and η-reductions. We consider some representative cases:

(i) For the case of $(\rightarrow\beta, \times\delta)$:

$$\begin{aligned} L((\lambda x:A\cdot M)N) &\twoheadrightarrow_\delta (L(\lambda x:A\cdot M))N \\ &\twoheadrightarrow_\delta (\lambda x:A\cdot LM)N \\ &\twoheadrightarrow_\beta \{N/x\}(LM) \\ &= L(\{N/x\}M) \\ &\twoheadleftarrow_\beta L((\lambda x:A\cdot M)N). \end{aligned}$$

(ii) For the case of $(\rightarrow\eta, \times\delta)$:

$$\begin{aligned} L(\lambda x:A\cdot Mx) &\twoheadrightarrow_\delta (\lambda x:A\cdot L(Mx)) \\ &\twoheadrightarrow_\delta \lambda x:A\cdot LMx \\ &\twoheadrightarrow_\eta LM \\ &\twoheadleftarrow_\eta L(\lambda x:A\cdot Mx). \end{aligned}$$

(iii) For the case of $(\times\eta, \times\delta)$:

$$\begin{aligned} &((\mathbf{1}L)(MN)[\overline{A_m}]\overline{N_n},\ (\mathbf{2}L)(MN)[\overline{A_m}]\overline{N_n}) \\ &\quad\twoheadrightarrow_\delta ((\mathbf{1}L)MN[\overline{A_m}]\overline{N_n},\ (\mathbf{2}L)(MN)[\overline{A_m}]\overline{N_n}) \\ &\quad\twoheadrightarrow_\delta ((\mathbf{1}L)MN[\overline{A_m}]\overline{N_n},\ (\mathbf{2}L)MN[\overline{A_m}]\overline{N_n}) \\ &\quad\twoheadrightarrow_\eta LMN[\overline{A_m}]\overline{N_n} \\ &\quad\twoheadleftarrow_\delta L(MN)[\overline{A_m}]\overline{N_n} \\ &\quad\twoheadleftarrow_\eta ((\mathbf{1}L)(MN)[\overline{A_m}]\overline{N_n},\ (\mathbf{2}L)(MN)[\overline{A_m}]\overline{N_n}) \end{aligned}$$

$$\begin{aligned} &((\mathbf{1}L)(\lambda x:A\cdot M)[\overline{A_m}]N\overline{N_n},\ (\mathbf{2}L)(\lambda x:A\cdot M)[\overline{A_m}]N\overline{N_n}) \\ &\quad\twoheadrightarrow_\delta ((\lambda x:A\cdot(\mathbf{1}L)M)[\overline{A_m}]N\overline{N_n}, (\lambda x:A\cdot(\mathbf{2}L)M)[\overline{A_m}]N\overline{N_n}) \\ &\quad\twoheadrightarrow_\beta (\{N/x\}((\mathbf{1}L)M)[\overline{A_m}]\overline{N_n},\ \{N/x\}((\mathbf{2}L)M)[\overline{A_m}]\overline{N_n}) \end{aligned}$$

$$
\begin{aligned}
&= ((\mathbf{1}L)(\{N/x\}M)[\overline{A_m}]\overline{N_n},\ (\mathbf{2}L)(\{N/x\}M)[\overline{A_m}]\overline{N_n}) \\
&\twoheadrightarrow_\eta L(\{N/x\}M)[\overline{A_m}]\overline{N_n} \\
&\twoheadleftarrow_\beta L(\lambda x{:}A \cdot M)[\overline{A_m}]N\overline{N_n} \\
&\twoheadleftarrow_\eta ((\mathbf{1}L)(\lambda x{:}A \cdot M)[\overline{A_m}]N\overline{N_n}, (\mathbf{2}L)(\lambda x{:}A \cdot M)[\overline{A_m}]N\overline{N_n})
\end{aligned}
$$

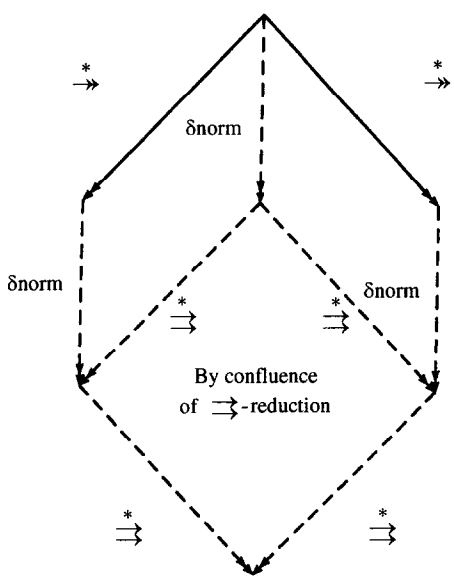
where in the second case the existence of the β-redex is guaranteed by the type hypothesis (which requires the term to have product type). □

Lemma 8. *$\rightrightarrows$ is confluent and terminating.*

The verification uses Girard's method of "candidats de reducibilité" [10, 8]. We omit the details, since the approach is essentially the same as that described by Gallier [8], defining a typed notion of "Girard sets" [8, p. 70], and verifying that Girard sets are candidates of reducibility, and that the type-indexed families of sets of strong normalizing terms and the families of sets of confluent terms both denote families of Girard sets. The consideration of terms in δ-normal form is crucial to use the approach described by Gallier [8], in particular in verifying the closure of Girard sets under the semantic type constructors [8, p. 71]. Finally we have:

Lemma 9. *$\beta\eta\delta$-reduction is canonical i.e., terminating and confluent.*

Proof. Termination follows from termination for $\rightrightarrows$, and the fact that $\twoheadrightarrow$-reduction sequences may be simulated by $\rightrightarrows$-reduction sequences. Confluence essentially follows a similar approach, verifying that the following diagram commutes (essentially using Hardin's interpretation technique [1]). □



Confluence for $\beta\eta\delta$-reduction provides an algorithm for converting terms and types to product normal form. We consider the following strategy in the next section for specifying extended patterns. We first of all consider terms and types in $\beta\eta$-normal form. This corresponds to converting projectors fst and snd to selectors **1** and **2**. This normal form corresponds to how we expect to enter terms in our calculus (for example using it as a metalanguage). We provide a statement of the extended pattern restriction for terms in $\beta\eta$-normal form. We then consider the simplification of the extended pattern restriction for terms in $\beta\eta\delta$-normal form i.e. in product normal form. Having described the extended pattern restriction for terms in $\beta\eta$-normal form, for the remainder of the paper we only consider terms in $\beta\eta\delta$-normal form.

3. Extended patterns

In this section we present our extension of patterns for product types. To specify the restrictions on the application of free variables, we reformulate the type rules to specify the restrictions for terms in normal form, and also incorporate a specification of which variables are "bound" (i.e. locally bound by a λ-abstraction) or free. To accomplish the latter we introduce a modified form of sequent with *quantifier contexts* rather than type environments:

$$\mathcal{Q} ::= \forall \overline{t_{k_1}} : \overline{K_{k_1}} \cdot \forall \overline{x_{k_2}} : \overline{A_{k_2}} \cdot \exists \overline{t'_{m_1}} : \overline{K'_{m_1}} \cdot \exists \overline{x'_{m_2}} : \overline{A'_{m_2}} \cdot \forall \overline{t''_{n_1}} : \overline{K''_{n_1}} \cdot \forall \overline{x''_{n_2}} : \overline{A''_{n_2}}.$$

We assume that all variables in a quantifier context are distinct. $\mathcal{Q}\exists t : K$ and $\mathcal{Q}\exists x : A$ denote the addition of the corresponding existential quantifier to the quantifier context. $\mathcal{Q}\forall t : K$ and $\mathcal{Q}\forall x : A$ denote the addition of the corresponding universal quantifier to the suffix of the equantifier context. All of these operations assume that the variable being added is not already bound in the quantifier context (as usual). A universally quantified variable is one which is locally bound (or *rigid*), while an existentially quantified variable is one which is free (or *flexible*). We will sometimes treat a quantifier context as a partial function: $(\mathcal{Q}\Xi x : A \cdot \mathcal{Q}')(x) = \Xi x : A$ for $\Xi \in \{\forall, \exists\}$, $dom(\mathcal{Q})$ denotes the set of variables bound in $\mathcal{Q}$, while $\mathcal{Q}_x$ denotes the restriction of $\mathcal{Q}$ to the variables bound to the left of x in $\mathcal{Q}$; similarly for type variables t.

For brevity, we assume that the constants in the signature Σ occur as an initial prefix of universally quantified variables in any such quantifier context. The existentially quantified variables correspond to the "free" variables in the terms. The variables which are universally quantified in the suffix of the context correspond to the locally bound variables, introduced by λ-abstraction.

The well-typed terms and types in $\beta\eta$-normal form (not necessarily product normal form) are a subset of those formed by the following grammar:

$$M_{\text{atom}} \in \text{Atomic Terms} ::= x \mid (L\, M_{\text{atom}}) \mid (M_{\text{atom}} M) \mid M_{\text{atom}}[A]$$
$$M \in \text{Compound Terms} ::= M_{\text{atom}} \mid \lambda x : A \cdot M \mid \Lambda t : K \cdot M \mid (M_1, M_2).$$

We make use of the following metafunctions defined over atomic well-typed terms and well-kinded types in $\beta\eta$-normal form:

$$\texttt{source}(x) \stackrel{\text{def}}{=} x \qquad \texttt{args}(x) \stackrel{\text{def}}{=} \{\ \}$$

$$\texttt{source}(L\,M) \stackrel{\text{def}}{=} \texttt{source}(M) \qquad \texttt{args}(L\,M) \stackrel{\text{def}}{=} \texttt{args}(M)$$

$$\texttt{source}(M\,[A]) \stackrel{\text{def}}{=} \texttt{source}(M) \qquad \texttt{args}(M\,[A]) \stackrel{\text{def}}{=} \texttt{args}(M) \cup \{A\}$$

$$\texttt{source}(M\,N) \stackrel{\text{def}}{=} \texttt{source}(M) \qquad \texttt{args}(M\,N) \stackrel{\text{def}}{=} \texttt{args}(M) \cup \{N\}.$$

Here $\texttt{source}(M)$ ($\texttt{source}(A)$) yields the variable at the head of a term M (type A). $\texttt{args}(M)$ ($\texttt{args}(A)$) yields the set of the arguments in the application subterms (subtypes) of $M(A)$, following a path from the root and through the rator part of any application. In the following, paramSet distributes the params metafunction over such a set of terms. params is necessary for the statement of the extended pattern restriction, to decompose pairs in operand positions into their components.

$$\texttt{paramSet}(\{M_i\}_i \cup \{A_j\}_j) \stackrel{\text{def}}{=} \left(\bigcup_i \texttt{params}(M_i)\right) \cup \{A_j\}_j$$

$$\texttt{params}((M, N)) \stackrel{\text{def}}{=} \texttt{params}(M) \cup \texttt{params}(N)$$

$$\texttt{params}(M) \stackrel{\text{def}}{=} \{M\} \text{ if } M \text{ is not a pair.}$$

We use these metafunctions to define the following predicate which restricts the well-typed normal form terms:

Definition 10. Given terms M and N in $\beta\eta$-normal form, define the predicate $\texttt{WF}_{\mathcal{Q}}(M, N)$ to be true if and only if

$$\mathcal{Q} = \forall \overline{t_{k_1} : K_{k_1}} \cdot \forall \overline{x_{k_2} : A_{k_2}} \cdot \exists \overline{t'_{m_1} : K'_{m_1}} \cdot \exists \overline{x'_{m_2} : A'_{m_2}} \cdot \forall \overline{t''_{n_1} : K''_{n_1}} \cdot \forall \overline{x''_{n_2} : A''_{n_2}}$$

and

(i) $\texttt{source}(M) \in \{x_i\}_i \cup \{x''_j\}_j$, or

(ii) $\texttt{source}(M) \in \{x'_i\}_i$ and $N \equiv L\,y$, some $y \in \{x''_j\}_j$, and $(L'\,y) \in \texttt{paramSet}(\texttt{args}(M))$ implies $L \not\leqslant L'$, $L' \not\leqslant L$.

$\texttt{WF}_{\mathcal{Q}}(M, A)$ is defined in a similar way. $\texttt{WF}_{\mathcal{Q}}(A, B)$ is the usual statement of the pattern restriction for simply-typed λ-terms.

Fig. 5 gives the rules which restrict normal form terms to extended patterns. These rules make use of an initial quantifier context, which is subsequently extended with local bindings. The importance of these rules is that they provide the following form of patterns for our language.

Definition 11 (*Extended patterns*). Given a quantifier context $\mathcal{Q}$, a term M in product normal form satisfies the (*simple*) *extended pattern restriction* if:

(i) $M \equiv \lambda x : A \cdot N$ and N satisfies the simple extended pattern restriction in $\mathcal{Q} \forall x : A$.

$$\textsc{Var}\quad \frac{(t \in dom(\mathcal{Q}))}{\mathcal{Q} \rhd_\Sigma t\ \mathbf{pat}} \qquad \frac{(x \in dom(\mathcal{Q}))}{\mathcal{Q} \rhd_\Sigma x\ \mathbf{pat}}$$

$$\textsc{Type}\quad \frac{}{\mathcal{Q} \rhd_\Sigma \mathsf{Type}\ \mathbf{pat}}$$

$$\Delta F,\ \Delta I\quad \frac{\mathcal{Q}\forall t : K \rhd_\Sigma A\ \mathbf{pat}}{\mathcal{Q} \rhd_\Sigma \Lambda t : K \cdot A\ \mathbf{pat}} \qquad \frac{\mathcal{Q}\forall t : K \rhd_\Sigma M\ \mathbf{pat}}{\mathcal{Q} \rhd_\Sigma \Lambda t : K \cdot M\ \mathbf{pat}}$$

$$\Delta E\quad \frac{\mathcal{Q} \rhd_\Sigma M\ \mathbf{pat} \quad \mathcal{Q} \rhd_\Sigma B\ \mathbf{pat}\ \mathrm{WF}_{\mathcal{Q}}(M, B)}{\mathcal{Q} \rhd_\Sigma M[B]\ \mathbf{pat}}$$

$$\to F\quad \frac{\mathcal{Q} \rhd_\Sigma A\ \mathbf{pat} \quad \mathcal{Q} \rhd_\Sigma B\ \mathbf{pat}}{\mathcal{Q} \rhd_\Sigma A \to B\ \mathbf{pat}} \qquad \frac{\mathcal{Q} \rhd_\Sigma K\ \mathbf{pat} \quad \mathcal{Q} \rhd_\Sigma K'\ \mathbf{pat}}{\mathcal{Q} \rhd_\Sigma K \to K'\ \mathbf{pat}}$$

$$\to I\quad \frac{\mathcal{Q}\forall x : A \rhd_\Sigma M\ \mathbf{pat}}{\mathcal{Q} \rhd_\Sigma \lambda x : A \cdot M\ \mathbf{pat}} \qquad \frac{\mathcal{Q}\forall t : K \rhd_\Sigma A\ \mathbf{pat}}{\mathcal{Q} \rhd_\Sigma \lambda t : K \cdot A\ \mathbf{pat}}$$

$$\to E\quad \frac{\mathcal{Q} \rhd_\Sigma M\ \mathbf{pat} \quad \mathcal{Q} \rhd_\Sigma N\ \mathbf{pat} \quad \mathrm{WF}_{\mathcal{Q}}(M, N)}{\mathcal{Q} \rhd_\Sigma (M\,N)\ \mathbf{pat}}$$

$$\to E\quad \frac{\mathcal{Q} \rhd_\Sigma A\ \mathbf{pat} \quad \mathcal{Q} \rhd_\Sigma B\ \mathbf{pat} \quad \mathrm{WF}_{\mathcal{Q}}(A, B)}{\mathcal{Q} \rhd_\Sigma (A\,B)\ \mathbf{pat}}$$

$$\times F\quad \frac{\mathcal{Q} \rhd_\Sigma A\ \mathbf{pat} \quad \mathcal{Q} \rhd_\Sigma B\ \mathbf{pat}}{\mathcal{Q} \rhd_\Sigma A \times B\ \mathbf{pat}}$$

$$\times I,\ \times E\quad \frac{\mathcal{Q} \rhd_\Sigma M\ \mathbf{pat} \quad \mathcal{Q} \rhd_\Sigma N\ \mathbf{pat}}{\mathcal{Q} \rhd_\Sigma (M, N)\ \mathbf{pat}} \qquad \frac{\mathcal{Q} \rhd_\Sigma M\ \mathbf{pat}}{\mathcal{Q} \rhd_\Sigma (LM)\ \mathbf{pat}}$$

Fig. 5. Extended pattern restrictions for terms in $\beta\eta$-normal form.

(ii) $M \equiv \Lambda t : K \cdot N$ and N satisfies the simple extended pattern restriction in $\mathcal{Q}\forall t : K$.
(iii) $M \equiv (N_1, N_2)$ and N_1 and N_2 satisfy the simple extended pattern restriction in $\mathcal{Q}$.
(iv) $M \equiv L\,x[\overline{A_m}]\overline{M_n}$ and each A_i and M_j satisfies the simple extended pattern restriction in $\mathcal{Q}$; and furthermore either x is a constant or a variable universally quantified in $\mathcal{Q}$, or x is existentially quantified in $\mathcal{Q}$ and
  (a) each $A_k \equiv t_k$, where t_k is universally quantified to the right of x in $\mathcal{Q}$; and if $t_{k_1} = t_{k_2}$ then $k_1 = k_2$ and
  (b) each $M_k = L'_k\,x_k$, where x_k is universally quanitified to the right of x in $\mathcal{Q}$; and if $x_{k_1} = x_{k_2}$ then $L_{k_1} \not\leqslant L_{k_2}, L_{k_2} \not\leqslant L_{k_1}$.

A term or type satisfying the pattern restriction in Fig. 5 may be converted to one satisfying this simpler statement of the restriction using the following currying

transformation. For any existentially quantified variable F of n term parameters (with formal parameter types $A_1,\ldots,A_n$), with the ith parameter of product type, replace all occurrences of F with the term $\lambda\overline{x_n}:\overline{A_n}\cdot Gx_1\ldots x_{i-1}\ (\mathbf{1}\ x_i)\ (\mathbf{2}\ x_i)x_{i+1}\ldots x_n$, where G is a new existentially quantified variable of the appropriate type. For example, given the term:

$$\Lambda t\cdot\lambda x\cdot\lambda y\cdot\lambda z\cdot F[t](\mathbf{1}\ x)((y,\mathbf{2}\ x),z)$$

the algorithm translates this to:

$$\Lambda t\cdot\lambda x\cdot\lambda y\cdot\lambda z\cdot G[t](\mathbf{1}\ x)\,y(\mathbf{2}\ x)z$$

via the substitution:

$$F\mapsto\Lambda t\cdot\lambda u\cdot\lambda v\cdot G[t]u(\mathbf{1}\ \mathbf{1}\ v)(\mathbf{2}\ \mathbf{1}\ v)(\mathbf{2}\ v).$$

This transformation will be shown to preserve the set of substitutions satisfying the original equality constraints.

Miller's pattern restriction [14] requires applications of "free" function variables to be η-convertible to the form $(F\overline{x_n})$ where the x_i's are distinct local constants introduced within the scope of F. Our pattern restriction generalizes applications of this form in two ways:

(i) we allow the head of such an application to be a locator applied to the free variable F; and
(ii) we allow the actual parameters to be locators applied to local constants (introduced within the scope of F), where moreover a local constant may be repeated within the argument list provided the locators applied to the two occurrences are not prefixes of each other.

Miller noted that with his restrictions the unification algorithm only needed to consider a restriction form of β-conversion which amounted to permutation of bound variables. Consideration of the β-reduction relation for the calculus with locators gives the following generalization of β_0-reduction:

Definition 12 ($\beta_0\pi$-*reduction*). Define $\beta_0\pi$-reduction to be the closure under the typed term formation rules of the following rewrite rules:

(i) $(\lambda t:K\cdot A)t'\twoheadrightarrow_{\beta_0}\{t'/t\}A$
(ii) $(\Lambda t:K\cdot M)[t']\twoheadrightarrow_{\beta_0}\{t'/t\}M$
(iii) $(\lambda x:A\cdot M)(Lw)\twoheadrightarrow_{\beta_0}\{(Lw)/x\}M$
(iv) $(L\,i)(M_1,M_2)\twoheadrightarrow_{\beta_0}L\,M_i$
(v) $L_1(L_2\,x[\overline{A_m}]\overline{M_n})\twoheadrightarrow_{\beta_0}(L_1\,L_2)\,x[\overline{A_m}]\overline{M_n}$
(vi) $L(\Lambda\overline{t_k}:\overline{K_k}\cdot\lambda\overline{x_l}:\overline{A_l}\cdot M)\twoheadrightarrow_{\beta_0}\Lambda\overline{t_k}:\overline{K_k}\cdot\lambda\overline{x_l}:\overline{A_l}\cdot L\,M.$

These rules simply correspond to combining the rules for $\beta\delta$-reduction which are applicable to terms and types resulting from substituting $\beta\delta$-normal forms for variables satisfying the extended pattern restriction. The following is verified by a tedious induction on reduction sequences.

Lemma 13. *If* $\Gamma \vdash_\Sigma M \in A$, $\mathcal{Q} \triangleright_\Sigma M$ **pat**, $\Gamma \vdash_\Sigma N \in B$ *and* $\mathcal{Q} \triangleright_\Sigma N$ **pat**, *where* Γ *equals* $\mathcal{Q}$ *with all quantifiers removed, and where* x *is existentially quantified with type* B *in* $\mathcal{Q}$. *Then* $\{N/x\}M \twoheadrightarrow_{\beta_0} M'$ *if and only if* $\{N/x\}M \twoheadrightarrow^*_{\beta\delta} M'$.

We see that $\beta_0\pi$-reduction is a restricted case of $\beta\delta$-reduction which is complete for extended patterns. Lemma 7 verifies that δ-reductions may be permuted with β-reductions. So $\beta_0\pi$-reduction can be viewed as β-reduction followed by transformation to product normal form.

4. The unification algorithm

In this section we present the unification algorithm, in the style of structural operational semantics. *Configurations* of the algorithm are triples of the form $(\mathcal{Q}; \mathscr{C}; \mathscr{C}_A)$, where:

$$\mathcal{Q}_G \in \text{Global Quantifier Context} ::= \forall \overline{t_{k_1}} : \overline{K_{k_1}} \cdot \forall \overline{x_{k_2}} : \overline{A_{k_2}} \cdot \exists \overline{t'_{m_1}} : \overline{K'_{m_1}} \cdot \exists \overline{x'_{m_2}} : \overline{A'_{m_2}}$$

$$\mathcal{Q}_L \in \text{Local Quantifier Context} ::= \forall \overline{t_{k_1}} : \overline{K_{k_1}} \cdot \forall \overline{x_{k_2}} : \overline{A_{k_2}}$$

$$\mathscr{C} \in \text{Constraint List} ::= \top \mid \mathscr{C}_1, \mathscr{C}_2 \mid \mathcal{Q}_L \cdot M = N \mid \mathcal{Q}_L \cdot A = B.$$

A constraint $M = N$ or $A = B$ is to be considered as a set rather than an ordered pair; this allows us to omit the symmetric statement of many of the rules of the algorithm. A configuration $(\mathcal{Q}; \mathscr{C}; \mathscr{C}_A)$ carries two constraint lists, a *list* of constraints $\mathscr{C}$ remaining to be analysed by the unification algorithm, and a *set* of *answer constraints* $\mathscr{C}_A$ which constitute the output of a successful execution of the algorithm. For each constraint $\mathcal{Q} \cdot M = N$ or $\mathcal{Q} \cdot A = B$ in $\mathscr{C}$ or $\mathscr{C}_A$, $\mathcal{Q}$ denotes the *local quantifier context*. The quantifier context $\mathcal{Q}$ in a configuration $(\mathcal{Q}; \mathscr{C}; \mathscr{C}_A)$ denotes the *global quantifier context*. Essentially each combination of the global quantifier context and each local quantifier context gives rise to a quantifier context in the sense described in the previous section; each constraint in a configuration has its own quantifier context, formed by the concatenation of the global and local quantifier contexts.

The extension of FV to quantifier contexts and constraint lists is unsurprising, for example $FV(\mathcal{Q}\forall x : A) = (FV(\mathcal{Q}) \cup FV(A)) - \{x\}$. The constraint list $\mathscr{C}_A$ is in *solved form* if $\mathscr{C}_A = ((t_i = A_i)_i, (x_j = M_i)_j)$ and for all i, t_i is existentially quantified in $\mathcal{Q}$, $t_i \notin (\bigcup_j FV(A_j)) \cup (\bigcup_k FV(M_k)) \cup FV(\mathcal{Q}) \cup FV(\mathscr{C})$, and $i \neq j$ implies $t_i \neq t_j$; similarly for the x_j's. Given an initial configuration with $\mathscr{C}_A$ empty, the algorithm maintains this invariant when adding answer constraints to $\mathscr{C}_A$.

Given configurations $(\mathcal{Q}; \mathscr{C}; \mathscr{C}_A)$ and $(\mathcal{Q}'; \mathscr{C}'; \mathscr{C}'_A)$, we denote that the latter is obtainable by applying a transition of the unification algorithm by $(\mathcal{Q}; \mathscr{C}; \mathscr{C}_A) \Rightarrow (\mathcal{Q}'; \mathscr{C}'; \mathscr{C}'_A)$; $\stackrel{*}{\Rightarrow}$ denotes the reflexive transitive closure of this relation. We postpone a formalization of well-formed configurations until we introduce the unification logic, in Section 6. A *final configuration* is one of the form $(\mathcal{Q}; \top; \mathscr{C}_A)$, where $\mathscr{C}_A$ is in solved form.

The algorithm is specified relative to a signature Σ which contains the primitive type constructors $\rightarrow$, $\times$ and Δ.

As with several formulations of β_0-unification, it is very important that the constraints being analysed be treated as a list rather than a multiset. As pointed out by Nipkow [20, 21], the algorithm may not terminate (due to a failure of the occurs check) if subconstraints introduced by a flexible–rigid step (the **Flex-Pair**, **Flex-Const** and **Flex-Param** rules below) are not analysed before any further constraints are considered. Consider for example $F = \mathsf{c}\, G$, $G = \mathsf{c}\, F$.

To shorten the statement of the algorithm, we use x, y, t to denote constants as well as variables, unless we explicitly state otherwise. If the algorithm terminates with a valid final configuration, then we say that the equated terms are *unifiable*, and a unifying substitution may be constructed from the answer constraint list. In the next section we show that, starting from a valid initial configuration, any execution of the unification algorithm is terminating. Furthermore, each step of the algorithm preserves the set of unifying substitutions for the constraints. Thus if the equated terms are unifiable, the substitution derived from the final answer constraint list is a most general unifier, with other unifiers derivable via composition with this substitution.

4.1. Rigid–rigid transitions

We begin with the rules for simplifying terms. The **Lam-Lam** and **Lam-Atom** rules remove λ-abstractions. The **Lam-Atom** rules perform implicit η-expansions of the atom on the right-hand side.

Lam-Lam-Ty $(\mathcal{Q};\ (\mathcal{Q}' \cdot \Lambda t : K \cdot M = \Lambda t : K \cdot N), \mathscr{C}; \mathscr{C}_A) \Rightarrow$

$(\mathcal{Q};\ (\mathcal{Q}' \forall t : K \cdot M = N), \mathscr{C}; \mathscr{C}_A).$

Lam-Lam-Tm $(\mathcal{Q};\ (\mathcal{Q}' \cdot \lambda x : A \cdot M = \lambda x : A \cdot N), \mathscr{C}; \mathscr{C}_A) \Rightarrow$

$(\mathcal{Q};\ (\mathcal{Q}' \forall x : A \cdot M = N), \mathscr{C}; \mathscr{C}_A).$

Lam-Atom-Ty $(\mathcal{Q};\ (\mathcal{Q}' \cdot (\Lambda t : K \cdot M) = (L\, x[\overline{A}]\overline{N})), \mathscr{C}; \mathscr{C}_A) \Rightarrow$

$(\mathcal{Q};\ (\mathcal{Q}' \forall t : K \cdot M = (L\, x[\overline{A}\ t]\overline{N})), \mathscr{C}; \mathscr{C}_A).$

Lam-Atom-Tm $(\mathcal{Q};\ (\mathcal{Q}' \cdot (\lambda x : A' \cdot M) = (L\, x[\overline{A}]\overline{N})), \mathscr{C}; \mathscr{C}_A) \Rightarrow$

$(\mathcal{Q};\ (\mathcal{Q}' \forall x : A \cdot M = (L\, x[\overline{A}]\overline{N} x)), \mathscr{C}; \mathscr{C}_A).$

The **Pair-Pair** and **Pair-Atom** rules remove pair constructors, reducing equality constraints to constraints over their constituent parts. Again the **Pair-Atom** rule performs implicit η-expansions of the atom. The restriction of **Pair-Atom** to rigid atoms is necessary to ensure termination, and is the first indication that products introduce non-trivial complications when reasoning about termination.

Pair-Pair $(\mathcal{Q};\ (\mathcal{Q}' \cdot (M_1, N_1) = (M_2, N_2)), \mathscr{C}; \mathscr{C}_A) \Rightarrow$

$(\mathcal{Q};\ (\mathcal{Q}' \cdot M_1 = M_2), (\mathcal{Q}' \cdot N_1 = N_2), \mathscr{C}; \mathscr{C}_A).$

Pair-Atom $(\mathcal{Q};\ (\mathcal{Q}' \cdot (M_1, M_2) = (L\, x[\overline{A}]\overline{N})), \mathscr{C}; \mathscr{C}_A) \Rightarrow$
$(\mathcal{Q};\ (\mathcal{Q}' \cdot M_1 = ((\mathbf{1}\ L)\, x[\overline{A}]\overline{N})), (\mathcal{Q}' \cdot M_2 = ((\mathbf{2}\ L)\, x[\overline{A}]\overline{N})), \mathscr{C}; \mathscr{C}_A)$
where x is a constant in Σ or is universally quantified in $\mathcal{Q}\mathcal{Q}'$.

The **rigid–rigid** rule simplifies equality constraints between two terms where the heads of both are the same constant (either global or introduced locally); in this case both must have the same locator applied to each of their heads.

Rigid–rigid $(\mathcal{Q};\ (\mathcal{Q}' \cdot (L\, x[\overline{A_m}]\overline{M_{m'}}) = (L\, x[\overline{B_n}]\overline{N_{n'}})), \mathscr{C};\ \mathscr{C}_A) \Rightarrow$
$(\mathcal{Q};\ (\mathcal{Q}' \cdot A_i = B_i)_{i \in [m]}, (\mathcal{Q}' \cdot M_j = N_j)_{j \in [m']}, \mathscr{C};\ \mathscr{C}_A)$
where x is a constant in Σ or is universally quantified in $\mathcal{Q}\mathcal{Q}'$, and $m = n$ and $m' = n'$.

4.2. *Flexible–rigid rules*

We next consider the transitions which simplify the forms of terms with flexible heads. The following rule eliminates locators applied to the head of a flexible term, by introducing a pair of new flexible variables. Combined with the simplification rules above, this will allow us to assume that the flexible side of any constraint has the form $(F\ p_1 \ldots p_n)$ where each p_i is a located variable. The restriction of application of this rule to actual locator occurrences (rather than being driven by the type of F) is important for the termination proof in the next section.

Loc-Elim $(\mathcal{Q};\ (\mathcal{Q}' \cdot (L\, i) F = N), \mathscr{C};\ \mathscr{C}_A) \Rightarrow$
$(\mathcal{Q}'';\ (\mathcal{Q}' \cdot L\, G_i = \{M/F\}N), \{M/F\}\mathscr{C};\ F = M, \{M/F\}\mathscr{C}_A)$
where $\mathcal{Q} = \mathcal{Q}_1 \cdot \exists F : (\Lambda \overline{t} : \overline{K} \cdot \overline{A} \to B_1 \times B_2) \cdot \mathcal{Q}_2$
and $\mathcal{Q}'' = \mathcal{Q}_1 \cdot \exists F : \exists G_1 : (\Lambda \overline{t} : \overline{K} \cdot \overline{A} \to B_1) \cdot \exists G_2 : (\Lambda \overline{t} : \overline{K} \cdot \overline{A} \to B_1) \cdot \mathcal{Q}_2$
and $M = \Lambda \overline{t} : \overline{K} \cdot \lambda \overline{x} : \overline{A} \cdot (G_1[\overline{t}]\,\overline{x}, G_2[\overline{t}]\,\overline{x})$.

The following rule simplifies the argument list to a flexible variable, by eliminating pairs in that argument list (as shown by the example on p. 19).

CurryTm $(\mathcal{Q}_1;\ (\mathcal{Q}_2 \cdot (F[\overline{A_k}]\overline{M_m}(N_1, N_2)\overline{M'_n}) = M''), \mathscr{C};\ \mathscr{C}_A) \Rightarrow$
$(\mathcal{Q}'_1;\ (\mathcal{Q}_2 \cdot F[\overline{A_m}]\overline{M_m} N_1 N_2 \overline{M'_n} = \{N/F\}M''), \{N/F\}\mathscr{C}; F = N, \{N/F\}\mathscr{C}_A)$
where $\mathcal{Q}_1 = \mathcal{Q}_{1,1} \cdot \exists F : (\Lambda \overline{t_k} : \overline{K_k} \cdot \overline{A_m} \to (\overline{A_N} \to A_{N_1} \times A_{N_2}) \to \overline{A'_n} \to B) \cdot \mathcal{Q}_{1,2}$
and $N = \Lambda \overline{t_k} : \overline{K_k} \cdot \lambda \overline{y_m} : \overline{A_m} \cdot \lambda y : (\overline{A_N} \to A_{N_1} \times A_{N_2}) \cdot \lambda \overline{y'_n} : \overline{A'_n} \cdot G[\overline{t_k}]\,\overline{y_m}(\mathbf{1}\ y)(\mathbf{2}\ y)\,\overline{y'_n}$
and $\mathcal{Q}'_1 = \mathcal{Q}_{1,1} \cdot \exists F.$
$\exists G : (\Lambda \overline{t_k} : \overline{K_k} \cdot \overline{A_m} \to (\overline{A_N} \to A_{N_1}) \to (\overline{A_N} \to A_{N_2}) \to (\overline{A'_n} \to B) \cdot \mathcal{Q}_{1,2}.$

The following rule simplifies a constraint involving a flexible term and a pair. Note that this rule does not involve an occurs check for the variable being eliminated. We discuss the termination of the algorithm with this rule in the next section.

Flex-Pair $(\mathcal{Q}_1;\ (\mathcal{Q}_2 \cdot (F[\overline{p^t_m}]\,\overline{p^x_n}) = (M_1, M_2)), \mathscr{C};\ \mathscr{C}_A) \Rightarrow$
$(\mathcal{Q}'_1;\ (\mathcal{Q}_2 \cdot (G_i[\overline{p^t_m}]\,\overline{p^x_n}) = M_i)_{i \in [2]}, \{N/F\}\mathscr{C};\ F = N, \{N/F\}\mathscr{C}_A)$
where $\mathcal{Q}_1 = \mathcal{Q}_{1,1} \cdot \exists F : (\Lambda \overline{t_k} : \overline{K_k} \cdot \overline{A_m} \to B_1 \times B_2) \cdot \mathcal{Q}_2$

and $\mathcal{Q}_1' = \mathcal{Q}_{1,1} \cdot \exists F \cdot \exists G_1 : (\Delta \overline{t_k} : \overline{K_k} \cdot \overline{A_m} \to B_1) \cdot \exists G_2 : (\Delta \overline{t_k} : \overline{K_k} \cdot \overline{A_m} \to B_2) \cdot \mathcal{Q}_{1,2}$
and $N = \Lambda \overline{t_k} : \overline{K_k} \cdot \lambda \overline{x_m} : \overline{A_m} \cdot (G_1[\overline{t_k}]\overline{x_m}, G_2[\overline{t_k}]\overline{x_m})$.

The following rules replace a flexible variable with the rigid expression it is equated to. Which rule is applied depends on the form of the head of the rigid term. If this head is a constant c in the signature Σ, then we apply the **Flex-Const** step; in this case c cannot be obtained from F by projecting on one of its arguments. Matters become more interesting if the head is a variable z occurring in the local quantifier context. In this case z can only be obtained as a result of instantiating F by projecting on one of F's arguments in the process of β-reducing the result of instantiating F. Furthermore since F's arguments are located variables, there must be an argument of F whose source is z and whose locator is a suffix of the locator applied to z in the head of the rigid term. Our restrictions ensure that there is no more than one such argument to F; if such an argument exists, it is uniquely chosen in the application of the **Flex-Param** step.

Flex-Const

$$(\mathcal{Q};\ (\mathcal{Q}' \cdot (F[\overline{p_{m_1}^t}]\,\overline{p_{M_2}^x}) = (L\,\mathrm{c}\,[\overline{B_{n_1}}]\overline{M_{n_2}})), \mathscr{C};\ \mathscr{C}_A) \Rightarrow$$
$$(\mathcal{Q}'';\ (\mathcal{Q}' \cdot (T_i\,\overline{p_{m_1}^t}) = B_i)_{i\in[n_1]}, (\mathcal{Q}' \cdot (G_j[\overline{p_{m_1}^t}]\,\overline{p_{m_2}^x}) = M_j)_{j\in[n_2]}, \{N/F\}\mathscr{C};$$
$$F = N, \{N/F\}\mathscr{C}_A)$$

where $\mathcal{Q} = \mathcal{Q}_1 \cdot \exists F : (\Delta \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \overline{A_{m_2}} \to A) \cdot \mathcal{Q}_2$
and c is in Σ and $F \notin \bigcup_j FV(M_j)$
and $N = \Lambda \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda \overline{x_{m_2}} : \overline{A_{m_2}}$.
$L\,\mathrm{c}\,[(T_1\,\overline{t_{m_1}}) \ldots (T_{n_1}\,\overline{t_{m_1}})](G_1[\overline{t_{m_1}}]\overline{x_{m_2}}) \ldots (G_{n_2}[\overline{t_{m_1}}]\overline{x_{m_2}})$
and $\mathcal{Q}'' = \mathcal{Q}_1 \cdot \exists F \cdot \exists T_1 : (\overline{K_{m_1}} \to \tau(B_1)) \ldots \exists T_{n_1} : (\overline{K_{m_1}} \to \tau(B_{n_1}))$.
$\exists G_1 : (\Delta \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \overline{A_{m_2}} \to \tau(M_1)) \ldots \exists G_{n_2} : (\Delta \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \overline{A_{m_2}} \to \tau(M_{n_2}))$.
$\{N/F\}\mathcal{Q}_2$.

Flex-Param

$$(\mathcal{Q};\ (\mathcal{Q}' \cdot (F[\overline{p_{m_1}^t}]\,\overline{p_{m_2}^x}) = (L\ z[\overline{B_{n_1}}]\overline{M_{n_2}}), \mathscr{C};\ \mathscr{C}_A) \Rightarrow$$
$$(\mathcal{Q}'';\ (\mathcal{Q}' \cdot (T_i\,\overline{p_{m_1}^t}) = B_i)_{i\in[n_1]}, (\mathcal{Q}' \cdot (G_j[\overline{p_{m_1}^t}]\,\overline{p_{m_2}^x}) = M_j)_{j\in[n_2]}, \{N/F\}\mathscr{C};$$
$$F = N, \{N/F\}\mathscr{C}_A)$$

where $\mathcal{Q} = \mathcal{Q}_1 \cdot \exists F : (\Delta \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \overline{A_{m_2}} \to A) \cdot \mathcal{Q}_2$
and z is universally quantified in $\mathcal{Q}_2\mathcal{Q}'$ and $F \notin \bigcup_j FV(M_j)$
and $p_k^x = (L''\,z)$ and $L = L'L''$ for some $k \in [m_2]$
and $N = \Lambda \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda \overline{x_{m_2}} : \overline{A_{m_2}}$.
$L'\,x_k[(T_1\,\overline{t_{m_1}}) \ldots (T_{n_1}\,\overline{t_{m_1}})](G_1[\overline{t_{m_1}}]\overline{x_{m_2}}) \ldots (G_{n_2}[\overline{t_{m_1}}]\overline{x_{m_2}})$
and $\mathcal{Q}'' = \mathcal{Q}_1 \cdot \exists F \cdot \exists T_1 : (\overline{K_{m_1}} \to \tau(B_1)) \cdots \exists T_{n_1} : (\overline{K_{m_1}} \to \tau(B_{n_1}))$.
$\exists G_1 : (\Delta \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \overline{A_{m_2}} \to \tau(M_1)) \cdots \exists G_{n_2} : (\Delta \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \overline{A_{m_2}} \to \tau(M_{n_2}))$.
$\{N/F\}\mathcal{Q}_2$.

Both of these rules make use of the $\tau(M)$ and $\tau(A)$ functions for computing types and kinds, respectively, as defined in Theorem 1. The type context is left implicit for

brevity. The following example illustrates why it is necessary to compute types (where c is a constant, F a flexible variable):

$$\mathsf{c} : A \times (\Delta t : \mathsf{Type} \cdot t), F : B$$
$$F = (\mathbf{2}\ \mathsf{c})[\Delta t_1 : \mathsf{Type} \cdot \Delta t_2 : \mathsf{Type} \cdot t_1 \to t_2][A][B](\mathbf{1}\ \mathsf{c}).$$

On the other hand the algorithm does not require the types of the flexible variables to execute, so this expensive computing of types and kinds may be postponed until after unification has terminated, when "computed" λ-abstractions need to be output.

4.3. *Flexible–flexible rules*

The rules in this subsection simplify constraints involving two flexible terms. Again we only consider the case for term variables, that for type variables being similar but simpler. In the case of first order unification this involves simply replacing one variable with another. In the case of unification with patterns, the scope of the variables being unified must be suitably combined. Where two occurrences of the same variable are equated, the algorithm proceeds much as in Miller's algorithm. Where occurrences of two distinct variables are equated, the next step in the algorithm is somewhat more involved due to the presence of located variables in the arguments of the variables.

Flex-Flex-Same The terms being unified are flexible terms with the same head, say $F[\overline{p_m^t}]\overline{p_n^x}$ and $F[\overline{q_m^t}]\overline{q_n^x}$. In this case let $\{i_j\}_{j\in[m']} \subseteq [m]$ and $\{k_j\}_{j\in[n']}$ be the maximal such index sets such that (a) $m' \leqslant m$, $1 \leqslant i_1 < i_2 < \cdots < i_{m'} \leqslant m$ and $p_{i_j}^t = q_{i_j}^t$ for each $j \in [m']$; and (b) $n' \leqslant n, 1 \leqslant k_1 < k_2 \cdots < k_{n'} \leqslant n$ and $p_{i_j}^x = q_{i_j}^x$ for each $j \in [n']$. We introduce a new variable G which is substituted for F. Letting $N = \Lambda\overline{t_m} : \overline{K_m} \cdot \lambda\overline{x_n} : \overline{A_n} \cdot G[t_{i_1} \ldots t_{i_{m'}}]x_{k_1} \ldots x_{k_{n'}}$, and letting $B' = \Delta t_{i_1} : K_{i_1} \ldots \Delta t_{i_{m'}} : K_{i_{m'}} \cdot A_{k_1} \to \cdots \to A_{k_{n'}} \to A$, where F has type $B = \Delta\overline{t_m} : \overline{K_m} \cdot \overline{A_n} \to A$, we make the transition:

$$(\mathcal{Q}_1 \cdot \exists F : B \cdot \mathcal{Q}_2; (\mathcal{Q}' \cdot (F[\overline{p_m^t}]\overline{p_n^x}) = (F[\overline{q_m^t}]\overline{q_n^x})), \mathscr{C}; \mathscr{C}_A) \Rightarrow$$
$$(\mathcal{Q}''; \{N/F\}\mathscr{C}; F = N, \{N/F\}\mathscr{C}_A)$$
$$\text{where } \mathcal{Q}'' = \mathcal{Q}_1 \cdot \exists F : B \cdot \exists G : B' \cdot \mathcal{Q}_2.$$

Flex-Flex-Diff The terms being unified are flexible terms different heads. Suppose the equated terms are $(F_1[\overline{p_{m_1}^t}]\overline{p_{n_1}^x})$ and $(F_2[\overline{q_{m_2}^t}]\overline{q_{n_2}^x})$. Assume the types of F_1 and F_2 are $B_1 = \Delta t_{m_1} : \overline{K_{m_1}} \cdot \overline{A_{n_1}} \to A$ and $B_2 = \Delta\overline{t'_{m_2}} : \overline{K'_{m_2}} \cdot \overline{A'_{n_2}} \to A'$, respectively. We introduce a new variable G which is substituted for both F_1 and F_2.

Define $P = \{p_i^t\}_i \cap \{q_j^t\}_j$ and let $m = |P|$. Define the partial monomorphisms φ_t and ψ_t such that $\varphi_t : [m] \to [m_1], \psi_t : [m] \to [m_2]$, and $p_{\varphi_t(i)}^t = q_{\psi_t(i)}^t$ for $i = 1, \ldots, m$. Let $\tilde{\varphi}_t(i) = p_{\varphi_t(i)}^t$ and $\tilde{\psi}_t(i) = q_{\psi_t(i)}^t$. Finally, define t_k'' and K_k'' for $k \in [m]$ by

$$\langle t_k'', K_k'' \rangle \stackrel{\text{def}}{=} \langle t_{\varphi_t(k)}, K_{\varphi_t(k)} \rangle.$$

Note that $\langle t_{\varphi_t(k)}, K_{\varphi_t(k)} \rangle = \langle t'_{\psi_t(k)}, K_{\psi_t(k)'} \rangle$, so the choice is arbitrary.

The definitions of $\tilde{\varphi}_x, \tilde{\psi}_x$ and $\{A''_k\}_{k\in[n]}$ for term variables are somewhat more complicated, because of the extended pattern restriction. Define $P' = \{p_k \in \{p_i^x\}_i \mid \exists q_l \in \{q_j^x\}_j \cdot q_l \leqslant p_k\}, Q' = \{q_l \in \{q_j^x\}_j \mid \exists p_k \in \{p_i^x\}_i \cdot p_k \leqslant q_l\}$, and let $n = |P' \cup Q'|$. Define the partial monomorphisms φ_x and ψ_x such that:

$$\varphi_x : [m] \to [n_1] \times [n_2] \quad \text{and} \quad \{q_j^x \mid \exists k \in [m] \cdot \varphi_x(k) = \langle i, j\rangle, p_i^x \leqslant q_j^x\} = Q'$$
$$\psi_x : [m] \to [n_2] \times [n_1] \quad \text{and} \quad \{p_i^x \mid \exists k \in [m] \cdot \psi_x(k) = \langle j, i\rangle, q_j^x \leqslant p_i^x\} = P'.$$

$\langle i,j\rangle$ is meta-notation for a pair, with meta-operations $\mathbf{1}\langle i,j\rangle = i$ and $\mathbf{2}\langle i,j\rangle = j$. Define the functions $\tilde{\varphi}_x$ and $\tilde{\psi}_x$, total on $[n]$, by

$$\tilde{\varphi}_x(k) \stackrel{\text{def}}{=} \begin{cases} x_{\mathbf{2}(\psi_x(k))} & \text{if } k \notin dom(\varphi_x) \\ L\, x_{\mathbf{1}(\varphi_x(k))} & \text{otherwise, where } L\, p^x_{\mathbf{1}(\varphi_x(k))} = q^x_{\mathbf{2}(\varphi_x(k))} \end{cases}$$

$$\tilde{\psi}_x(k) \stackrel{\text{def}}{=} \begin{cases} x'_{\mathbf{2}(\varphi_x(k))} & \text{if } k \notin dom(\psi_x) \\ L\, x'_{\mathbf{1}(\psi_x(k))} & \text{otherwise, where } L\, p^x_{\mathbf{1}(\psi_x(k))} = q^x_{\mathbf{2}(\psi_x(k))}. \end{cases}$$

Finally, define A''_k for $k \in [n]$ by

$$A''_k \stackrel{\text{def}}{=} \begin{cases} A_{\mathbf{2}(\psi_x(k))} & \text{if } k \notin dom(\varphi_x) \\ A'_{\mathbf{2}(\varphi_x(k))} & \text{otherwise.} \end{cases}$$

Let $B = \Delta\overline{t''_m} : \overline{K''_m} \cdot \overline{A''_n} \to A$. Letting

$$M_1 = \Lambda\overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda\overline{x_{n_1}} : \overline{A_{n_1}} \cdot G[\tilde{\varphi}_t(1) \ldots \tilde{\varphi}_t(m)]\tilde{\varphi}_x(1) \cdots \tilde{\varphi}_x(n)$$
$$M_2 = \Lambda\overline{t'_{m_2}} : \overline{K'_{m_2}} \cdot \lambda\overline{x'_{n_2}} : \overline{A'_{n_2}} \cdot G[\tilde{\psi}_t(1) \ldots \tilde{\psi}_t(m)]\tilde{\psi}_x(1) \cdots \tilde{\psi}_x(n)$$

we make the transition:

$$(Q_1 \exists F_1 : B_1 \cdot Q_2 \exists F_2 : B_2 \cdot Q_3;\ (Q' \cdot (F_1[\overline{p^t_{m_1}}]\,\overline{p^x_{n_1}}) = (F_2[\overline{q^t_{m_2}}]\overline{q^x_{n_2}}), \mathscr{C}; \mathscr{C}_A)$$
$$\Rightarrow (Q''; \{M_1/F_1, M_2/F_2\}\mathscr{C}; F_1 = M_1, F_2 = M_2, \{M_1/F_1, M_2/F_2\}\mathscr{C}_A)$$

where $Q'' = Q_1 \exists G : B \cdot \exists F_1 : B_1 \cdot Q_2 \exists F_2 : B_2 \cdot Q_3)$.

To understand the **Flex-Flex-Diff** rule, consider the following constraint:

$$(F\ x(\mathbf{1}\ y)z\,w) = (G(\mathbf{1}\ x)\ (\mathbf{2}\ x)\ y\ z).$$

The algorithm computes the substitution:

$$F \mapsto \lambda\overline{x_4} : \overline{A_4} \cdot H\ x_2 x_3(\mathbf{1}\ x_1)\ (\mathbf{2}\ x_1)$$
$$G \mapsto \lambda\overline{x'_4} : \overline{A'_4} \cdot H\ (\mathbf{1}\ x'_3)\ x'_4 x'_1 x'_2$$

where H has the type $A_2 \to A'_4 \to A'_1 \to A'_2 \to B$ (where B is the common range type for $(F\ x(\mathbf{1}\ y)z\,w)$ and $(G(\mathbf{1}\ x)\ (\mathbf{2}\ x)\ y\ z)$).

An alternative formulation of the **Flex-Flex-Diff** rule is possible, in terms of one rule which is repeatedly applied to specialize the argument lists of the unificands,

followed by an application of a rule similar to that found in Miller's algorithm [14], which intersects the argument lists. Under this formulation, the example above would be repeatedly simplified as follows:

$$(F\ x\ (\mathbf{1}\ y)\ z\ w) = (G\ (\mathbf{1}\ x)\ (\mathbf{2}\ x)\ y\ z)$$
$$(F'\ (\mathbf{1}\ x)\ (\mathbf{2}\ x)\ (\mathbf{1}\ y)\ z\ w) = (G\ (\mathbf{1}\ x)\ (\mathbf{2}\ x)\ y\ z)$$
$$(F'\ (\mathbf{1}\ x)\ (\mathbf{2}\ x)\ (\mathbf{1}\ y)\ z\ w) = (G'\ (\mathbf{1}\ x)\ (\mathbf{2}\ x)\ (\mathbf{1}\ y)\ z)$$
$$(H\ (\mathbf{1}\ x)\ (\mathbf{2}\ x)\ (\mathbf{1}\ y)\ z) = (H\ (\mathbf{1}\ x)\ (\mathbf{2}\ x)\ (\mathbf{1}\ y)\ z).$$

4.4. Raising transitions

We have assumed a particularly restrictive form of quantifier context, of the form $\forall\exists\forall$. Logic programming in metalanguages such as L_λ [14] (that might be augmented with $\beta_0\pi$-unification) give rise to arbitrary interleavings of universal and existential quantifiers. The purpose of the raising transitions is to permute existential quantifiers with universal quantifiers binding local constants, raising all existentially quanitified variables to the global quantifier context. In this way arbitrary quantifier contexts may be transformed into the restrictive form of quantifier context we have been considering. In Section 6 we extend the Miller–Pfenning unification logic with a notion of well-typing for substitutions which allows us to reason about the correctness of the unification algorithm in the presence of raising transitions.

$$\mathbf{Raise}_{KT}\ (\mathcal{Q};\ (\mathcal{Q}'\forall t:K\exists F:A\mathcal{Q}''\cdot\mathscr{C}),\mathscr{C}';\mathscr{C}_A) \Rightarrow$$
$$(\mathcal{Q};(\mathcal{Q}'\exists F:(\Lambda t:K\cdot A)\cdot\forall t:K\mathcal{Q}''\cdot\{(F[t])/F\}\mathscr{C}),\mathscr{C}';\mathscr{C}_A)$$
$$\mathbf{Raise}_{TT}\ (\mathcal{Q};\ (\mathcal{Q}'\forall x:A\exists F:B\mathcal{Q}''\cdot\mathscr{C}),\mathscr{C}';\mathscr{C}_A) \Rightarrow$$
$$(\mathcal{Q};(\mathcal{Q}'\exists F:A\to B\forall x:A\mathcal{Q}''\cdot\{(F\ x)/F\}\mathscr{C}),\mathscr{C}';\mathscr{C}_A).$$

The following rules perform routine housekeeping. Combined with the raising rules, these allow us to maintain global quantifier context in the form $\exists\overline{t^1_{k^1_1}:K^1_{k^1_1}}\cdot\exists\overline{x^1_{k^1_2}:A^1_{k^1_2}}$, where each constraint has a local quantifier context of the form $\forall\overline{t^2_{k^2_1}:K^2_{k^2_1}}\cdot\forall\overline{x^2_{k^2_2}:A^2_{k^2_2}}$.

$$\mathbf{RaiseTm}\ (\mathcal{Q};\ (\exists x:A\mathcal{Q}'\cdot\mathscr{C}'),\mathscr{C};\mathscr{C}_A) \Rightarrow (\mathcal{Q}\exists x:A;\ (\mathcal{Q}'\cdot\mathscr{C}'),\mathscr{C};\mathscr{C}_A)$$
$$\mathbf{Permute}\ (\mathcal{Q};\ (\mathcal{Q}'\forall x:A\forall t:K\mathcal{Q}''\cdot\mathscr{C}),\mathscr{C}';\mathscr{C}_A) \Rightarrow$$
$$(\mathcal{Q};(\mathcal{Q}'\forall t:K\forall x:A\mathcal{Q}''\cdot\mathscr{C}),\mathscr{C}';\mathscr{C}_A).$$

In the latter rule, we assume that the type variable t has been renamed to avoid any accidental capture of free variables in the type A before any permutation has been performed.

5. Termination of the algorithm

In this section we reason about the termination of the algorithm provided in the previous section. We verify that the algorithm terminates on all inputs, in the next

section we verify that each transition of the algorithm preserves the set of solutions to the original unification algorithm. These results combined provide us with soundness and completeness results for the algorithm.

The termination of the algorithm is complicated by products. The rule for eliminating locators requires the introduction of (permanent) new variables (for the left and right components of pairs being projected out of). For example if we consider the constraints:

$$(\mathbf{2}\,\mathbf{2}\,\mathbf{2}F) = (\mathsf{c}\,G), \quad G = F$$

where for example F has type $A \equiv (\mathsf{int} \times (\mathsf{int} \times (\mathsf{int} \times \mathsf{int})))$ and c has type $A \to \mathsf{int}$, then repeated applications of **Loc-Elim**, followed by a final application of **Flex-Const** and **Flex-Flex-Diff**, reduce these to the single constraint

$$G = (H_{1,1}, (H_{2,1}, (H_{3,1}, (\mathsf{c}\,G)))).$$

Repeated applications of **Flex-Pair** and **Flex-Flex-Diff** lead to the final constraint

$$H_{3,2} = \mathsf{c}\,(H_{1,1}, (H_{2,1}, (H_{3,1}, H_{3,2})))$$

at which point the occurs check in **Flex-Const** rule fails. This example also illustrates that polymorphism introduces extra complications into reasoning about termination with products. We cannot determine in advance how many new variables are to be introduced by the locator elimination rule, on the basis of type information alone (since type variables may be instantiated with product types).

The intuition for the termination argument is as follows: The trees underlying the term calculus are in fact directed graphs, where variables (which can be thought of as pointers) give rise to shared vertices. Each variable vertex with a locator applied to it is the root of a directed binary tree, with edges labelled by selectors (selector edges). We will refer to this as a *locator tree*, while we will refer to a path made up of selector edges as a *locator path*. Such a graph also contains edges corresponding to equated vertices (constraint edges). The unification algorithm may be seen as merging vertices joined by constraint edges, in effect "pasting together" locator trees. The termination argument will be based on this intuitive explanation of the unification of located variables. In general every cycle will need to pass through a rigid term vertex, provided every constraint is well-typed. This is ensured by the unification algorithm, which implicitly builds such a graph. This will greatly simplify the termination argument, since it allows us to assume that the locator tree rooted at any variable vertex has finite depth. We will use locator occurrences to get a "true" estimate of the number of variable vertices in the set of locator trees implicitly constructed by unification.

The proof of termination is provided in Section 5.2. The difficult part of the proof is for the application of **Flex-Pair**. This part of the proof is therefore worked out in detail in Section 5.4. This part of the proof makes use of a *locator tree construction*, provided in Section 5.1, and a verification of termination for a particular class of constraints, provided in Section 5.3.

5.1. The locator tree construction

Definition 14 (*First-order reduction*). We assume given the initial "raw syntax" for types and terms provided in Section 2, but without the λ and Λ constructors. We assume that the original signature Σ is augmented with a single type constant tc_T and term constant c_V. We define the first-order reduction of terms and types in the original term calculus to terms in this subset calculus:

$$\mathscr{R}[\![\lambda t:K\cdot A]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \mathscr{R}[\![A]\!](\mathcal{Q}\forall t:K)$$

$$\mathscr{R}[\![(\mathsf{tc}\overline{A_m})]\!](\mathcal{Q}) \stackrel{\text{def}}{=} (\mathsf{tc})\overline{\mathscr{R}[\![A_m]\!](\mathcal{Q})}$$

$$\mathscr{R}[\![(t\overline{A_m})]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \begin{cases} \mathsf{tc}_T\,\overline{\mathscr{R}[\![A_m]\!](\mathcal{Q})} & \text{if } t \text{ is universally quantified in } \mathcal{Q} \\ t & \text{otherwise} \end{cases}$$

$$\mathscr{R}[\![\Lambda t:K\cdot M]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \mathscr{R}[\![M]\!](\mathcal{Q}\forall t:K)$$

$$\mathscr{R}[\![\lambda x:A\cdot M]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \mathscr{R}[\![M]\!](\mathcal{Q}\forall x:A)$$

$$\mathscr{R}[\![(M,N)]\!](\mathcal{Q}) \stackrel{\text{def}}{=} (\mathscr{R}[\![M]\!](\mathcal{Q}),\mathscr{R}[\![N]\!](\mathcal{Q}))$$

$$\mathscr{R}[\![L\,\mathsf{c}\overline{[A_m]}\,\overline{M_n}]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \mathsf{c}\,\overline{[\mathscr{R}[\![A_m]\!](\mathcal{Q})]}\,\overline{\mathscr{R}[\![M_n]\!](\mathcal{Q})}$$

$$\mathscr{R}[\![L\,x\overline{[A_m]}\,\overline{M_n}]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \begin{cases} \mathsf{c}_V\ \overline{[\mathscr{R}[\![A_m]\!](\mathcal{Q})]}\,\overline{\mathscr{R}[\![M_n]\!](\mathcal{Q})} \\ \quad \text{if } x \text{ is universally quantified in } \mathcal{Q} \\ L\,x \text{ otherwise.} \end{cases}$$

For an empty quantifier context, we denote the first-order reduction by $\mathscr{R}[\![A]\!](\mathscr{R}[\![M]\!])$. We extend this straightforwardly to a mapping from constraint lists to constraint multisets:

$$\mathscr{R}[\![\top]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \{\ \}$$

$$\mathscr{R}[\![\mathscr{C}_1,\mathscr{C}_2]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \mathscr{R}[\![\mathscr{C}_1]\!](\mathcal{Q}) \cup \mathscr{R}[\![\mathscr{C}_2]\!](\mathcal{Q})$$

$$\mathscr{R}[\![\Xi t:K\,\mathcal{Q}'\mathscr{C}]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \mathscr{R}[\![\mathcal{Q}'\mathscr{C}]\!](\mathcal{Q}\Xi t:K)$$

$$\mathscr{R}[\![\Xi x:A\,\mathcal{Q}'\mathscr{C}]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \mathscr{R}[\![\mathcal{Q}'\mathscr{C}]\!](\mathcal{Q}\Xi x:A)$$

$$\mathscr{R}[\![A=B]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \{\mathscr{R}[\![A]\!](\mathcal{Q}) = \mathscr{R}[\![B]\!](\mathcal{Q})\}$$

$$\mathscr{R}[\![M=N]\!](\mathcal{Q}) \stackrel{\text{def}}{=} \{\mathscr{R}[\![M]\!](\mathcal{Q}) = \mathscr{R}[\![N]\!](\mathcal{Q})\}.$$

The first-order reduction process produces a multiset of constraints between first-order trees, while retaining the rigid structure of the trees (replacing λ-bound variables by constants). Occurrences of located variables correspond to the addresses of vertices in the locator trees which are implicitly embedded in these trees.

Definition 15 (*Locator trees*). Assume given a set of located variables $\mathscr{V}$ satisfying the following properties:

Prefix closure: If $((iL)F) \in \mathscr{V}$ (for some $i \in [2]$), then $(LF) \in \mathscr{V}$.

Binary completeness: If $((\mathbf{1}L)F) \in \mathscr{V}$, then $((\mathbf{2}L)F) \in \mathscr{V}$, and vice versa.

Then define the *locator tree* $\mathscr{T}(\mathscr{V})$ to have:

(i) vertices $V(\mathscr{T}(\mathscr{V})) = \mathscr{V}$, and

(ii) edges $E(\mathscr{T}(\mathscr{V})) = \{((LF),((iL)F)) \mid (LF),((IL)F) \in \mathscr{V}\}$

Binary completeness ensures that (in the absence of any further identification) each vertex has exactly zero or two subgraphs. Prefix closure ensures that, for any vertex labelled with a located variable LF, there exists a path from a vertex labelled with F to LF.

Definition 16 (*Construction of locator tree*). Given a configuration quantifier context $\mathscr{Q}$ and a constraint list $\mathscr{C}$, construct a locator tree $\mathscr{L}(\mathscr{Q};\mathscr{C})$ as follows. Let $\mathscr{V}$ consist of the set of located variables $\{\{L_i F_i\}\}_i \cup \{L'_j T_j\}\}_j$ which occur in $\mathscr{R}[\![\mathscr{C}]\!](\mathscr{Q})$. Let $\mathscr{V}'$ result from exhaustively applying the following closure transitions to $\mathscr{V}$ (where $\cup$ is disjoint union):

$$\mathscr{V} \cup \{(iL)F\} \Rightarrow \mathscr{V} \cup \{(iL)F, LF\} \quad \text{where } (LF) \notin \mathscr{V}$$

$$\mathscr{V} \cup \{(\mathbf{1}L)F\} \Rightarrow \mathscr{V} \cup \{(\mathbf{1}L)F, (\mathbf{2}L)F\} \quad \text{where } ((\mathbf{2}L)F) \notin \mathscr{V}$$

$$\mathscr{V} \cup \{(\mathbf{2}L)F\} \Rightarrow \mathscr{V} \cup \{(\mathbf{2}L)F, (\mathbf{1}L)F\} \quad \text{where } ((\mathbf{1}L)F) \notin \mathscr{V}.$$

Finally define $\mathscr{L}(\mathscr{Q};\mathscr{C})$ to be $\mathscr{T}(\mathscr{V}')$. Given a configuration $(\mathscr{Q};\mathscr{C};\mathscr{C}_A)$, define $\mathscr{L}((\mathscr{Q};\mathscr{C};\mathscr{C}_A))$ to be $\mathscr{L}(\mathscr{Q};\mathscr{C})$.

We use the locator tree $\mathscr{L}((\mathscr{Q};\mathscr{C};\mathscr{C}_A))$ to define a measure of the number of vertices referenced by located variables in the constraints. Consider the following example:

$$LF = (L_G G, L_H H), \ldots, (L'_G\, 1\, L)F \ldots (L'_H\, 2\, L)F \ldots$$

$$\Rightarrow \cdots (L'_G L_G)G \cdots (L'_H L_H)H \cdots.$$

Fig. 6 shows the effect on locator trees of this transition. The **Loc-Elim** rule introduces new variables to name each of the vertices on the path from F to LF. Finally an application of **Flex-Pair** deletes the (renamed) vertex for LF and embeds the two subtrees into the subtrees of G and H. Throughout all of this the number of vertices in the set of locator trees is strictly reduced by each rule application.

Note that the locator tree construction does not give an accurate measure of the size of the binary tree rooted at a variable. For example, given the constraint list $(F = (M,N), C)$, where there are no located occurrences of F in $\mathscr{C}$, the locator tree rooted at F contains only the vertex for F itself. The termination measure only considers the "identifiable" vertices in the binary tree located at a variable. The sum of these measures, over all remaining variables, is verified to decrease, although as shown in

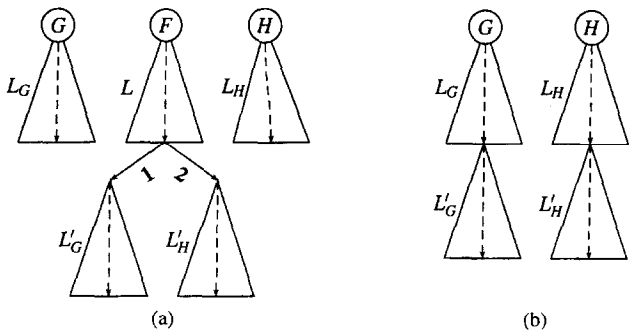


Fig. 6. Example of merged locators: $LF=(L_G\,G, L_H\,H),\ldots,(L'_G 1\,L)F\ldots(L'_H\,2\,L)F\ldots$

the previous example the count may increase for a particular variable because of the transplanting of subtrees.

The **Flex-Pair** rule does not require a check for any occurrences of the variable being eliminated in the right-hand side. The termination of the algorithm in the absence of this occurs check is moderately subtle. In the constraint $F=(M,N)$, located occurrences of F may occur in M and N without making the terms non-unifiable. Consider the following examples:

$$F=(\mathbf{2}F,\mathbf{1}F)\overset{\{(H_1,H_2)/F\}}{\Longrightarrow}H_1=H_2,\ H_2=H_1\overset{\{H_2/H_1\}}{\Longrightarrow}H_2=H_2$$

$$F=(\mathbf{1}F,\mathbf{1}F)\overset{\{(H_1,H_2)/F\}}{\Longrightarrow}H_1=H_1,\ H_2=H_1\ \Rightarrow\ H_2=H_1$$

$$F=(\mathsf{c}_1,\mathsf{c}_2(\mathbf{1}F))\overset{\{(H_1,H_2)/F\}}{\Longrightarrow}H_1=\mathsf{c}_1,H_2=\mathsf{c}_2\,H_1\overset{\{\mathsf{c}_1/H_1\}}{\Longrightarrow}H_2=\mathsf{c}_2\,\mathsf{c}_1.$$

Given the constraint $F=M$, define a *rigid self-reference* in M to be a located occurrence of F and M, LF, which is a subterm of a rigid application. Define a *recursive self-reference* in M to be a located occurrence of F in M, LF, where the locator L is a *strict* prefix of the locator path L' in M leading to this occurrence,[4] or vice versa, *including locator paths which chain through other located occurrences of* F. The extra chaining condition may appear odd; consider the example $F=(\mathbf{2}F,\mathsf{c}(\mathbf{1}F))$. The following table summarizes the combinations of rigid/non-rigid and recursive/non-recursive self-references in $F=M$:

	Recursive	Non-recursive
Rigid	$F=(\mathsf{c}_1,\mathsf{c}_2(\mathbf{2}F))$	$F=(\mathsf{c}_1,\mathsf{c}_2(\mathbf{1}F))$
Non-rigid	$F=(\mathbf{11}F,M)$	$F=(\mathbf{1}F,\mathbf{1}F)$

A pair can be considered as a binary tree constructor, with variables and non-pair term constructors at the leaves. Non-rigid non-recursive self-references in such a tree

[4] In other words, $L\equiv i_1\cdots i_m$ and $L'\equiv i_1\cdots i_n$, and $m<n$, $m\neq n$.

correspond to sharing of such subtrees, giving rise to a directed graph (digraph) structure. Provided all such non-rigid self-references are not recursive, the tree is acyclic. We will make essential use of the acyclicity of this digraph when verifying termination, by expanding the digraph out to the underlying tree, with shared subtrees duplicated, and counting the number of vertices.

A non-rigid recursive self-reference can lead to a non-terminating sequence of transitions, due to the omission of the occurs check in **Flex-Pair**:

$$F = (F, M) \overset{\{(H_1, H_2)/F\}}{\Longrightarrow} H_1 = (H_1, H_2),\ H_2 = M.$$

However such circularities are disallowed by the well-typedness of the constraints considered by the unification algorithm. The occurs check for rigid type expressions is essential to ensure this. Consider for example:

$$\exists A_1, B_1, A_2, B_2 : \mathsf{Type} \cdot \exists F : A_1 \times B_1 \cdot \exists G : A_2 \times B_2 \cdot$$
$$\mathsf{c}[A_1](\mathbf{1}\,F) = \mathsf{c}\,[A_2 \times B_2]\,G,\ \mathsf{c}\,[A_2](\mathbf{1}\,G) = \mathsf{c}[A_1 \times B_1]\,F$$

which establishes the cyclic locator path $\mathbf{1}\,F = G$, $\mathbf{1}\,G = F$ in the absence of the occurs check for the unified types. Because function variables may be polymorphic, the pattern restriction on function variable applications is also essential to disallow circularities; consider $\exists F : (\Delta t : \mathsf{Type} \cdot t) \cdot \mathsf{fst}(F[A \times B]) = F[A]$.

Similarly to non-rigid non-recursive self-references, rigid non-recursive self-references are relatively benign and allow subtrees to be shared within rigid subterms. Rigid recursive self-references introduce cycles into the term graph, and furthermore are not prevented by well-typing. In this case we rely on the occurs checks for the **Flex-Const** and **Flex-Param** rules to ensure termination. Consider the examples:

$$F = (\mathsf{c}_1, \mathsf{c}_2\,F) \overset{\{(H_1, H_2)/F\}}{\Longrightarrow} H_1 = \mathsf{c}_1, H_2 = \mathsf{c}_2(H_1, H_2) \overset{\{\mathsf{c}_1/H_1\}}{\Longrightarrow} H_2 = \mathsf{c}_2(\mathsf{c}_1, H_2)$$

$$F = (\mathsf{c}_1, \mathsf{c}_2(\mathbf{2}\,F)) \overset{\{(H_1, H_2)/F\}}{\Longrightarrow} H_1 = \mathsf{c}_1, H_2 = \mathsf{c}_2\,H_2 \overset{\{\mathsf{c}_1/H_1\}}{\Longrightarrow} H_2 = \mathsf{c}_2\,H_2$$

$$F = (\mathsf{c}_1, (\mathsf{c}_2, \mathsf{c}_3 F)) \overset{\{(H_1, H_2)/F\}}{\Longrightarrow} H_1 = \mathsf{c}_1,$$

$$H_2 = (\mathsf{c}_2, \mathsf{c}_3(H_1, H_2)) \overset{\{\mathsf{c}_1/H_1\}}{\Longrightarrow} H_2 = (\mathsf{c}_2, \mathsf{c}_3(\mathsf{c}_1, H_2)) \overset{\{(H_{2,1}, H_{2,2})/H_2\}}{\Longrightarrow} H_{2,1} = \mathsf{c}_2,$$

$$H_{2,2} = \mathsf{c}_3(\mathsf{c}_1, (H_{2,1}, H_{2,2})) \overset{\{\mathsf{c}_1/H_1\}}{\Longrightarrow} H_{2,2} = \mathsf{c}_3(\mathsf{c}_1, (\mathsf{c}_2, H_{2,2}))$$

$$F = (\mathbf{2}\,F, \mathsf{c}(\mathbf{1}\,F)) \overset{\{(H_1, H_2)/F\}}{\Longrightarrow} H_1 = H_2,\ H_2 = \mathsf{c}\,H_1 \overset{\{H_2/H_1\}}{\Longrightarrow} H_2 = \mathsf{c}\,H_2.$$

5.2. *Proof of termination*

Theorem 17 (Termination). *There is no infinite sequence of configuration transitions*

$$(\mathcal{Q}_1; \mathscr{C}_1; \mathscr{C}_{A_1}) \Rightarrow \cdots \Rightarrow (\mathcal{Q}_i; \mathscr{C}_i; \mathscr{C}_{A_i}) \Rightarrow (\mathcal{Q}_{i+1}; \mathscr{C}_{i+1}; \mathscr{C}_{A_{i+1}}) \Rightarrow \cdots.$$

Proof. We define the following well-founded measure and verify that it is decreasing on each transition of the unification algorithm. Define the size of a type or term as follows, with respect to a quantifier context $\mathcal{Q}$:

$$|\lambda\overline{t_m}:\overline{K_m}\cdot t'\,\overline{A_{m'}}|_{\mathcal{Q}} \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } t' \text{ is existentially quantified in } \mathcal{Q} \\ 1+m+\sum_{i=1}^{m'}|A_i|_{\mathcal{Q}} & \text{otherwise} \end{cases}$$

$$|\Lambda\overline{t_m}:\overline{K_m}\cdot\lambda\overline{x_n}:\overline{A_n}\cdot L\,z[\overline{A_{m'}}]\overline{M_{n'}}|_{\mathcal{Q}} \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } z \text{ is existentially quantified in } \mathcal{Q} \\ 1+m+n+(\sum_{i=1}^{m'}|A_i|_{\mathcal{Q}})+(\sum_{j=1}^{n'}|M_j|_{\mathcal{Q}}) & \\ & \text{otherwise} \end{cases}$$

$$|\Lambda\overline{t_m}:\overline{K_m}\cdot\lambda\overline{x_n}:\overline{A_n}\cdot(M,N)|_{\mathcal{Q}} \stackrel{\text{def}}{=} 1+m+n+|M|_{\mathcal{Q}}+|N|_{\mathcal{Q}}$$

$$|\mathcal{Q}'(A=B)|_{\mathcal{Q}} \stackrel{\text{def}}{=} \{1+|A|_{\mathcal{Q}\mathcal{Q}'}+|B|_{\mathcal{Q}\mathcal{Q}'}\}$$

$$|\mathcal{Q}'(M=N)|_{\mathcal{Q}} \stackrel{\text{def}}{=} \{1+|M|_{\mathcal{Q}\mathcal{Q}'}+|N|_{\mathcal{Q}\mathcal{Q}'}\}$$

$$|\mathscr{C}_1,\mathscr{C}_2|_{\mathcal{Q}} \stackrel{\text{def}}{=} \ |\ \mathscr{C}_1|_{\mathcal{Q}} \cup |C_2|_{\mathcal{Q}}$$

$$|\top|_{\mathcal{Q}} \stackrel{\text{def}}{=} \{\ \}$$

where union is understood to be multiset union. Define $TS((\mathcal{Q};\mathscr{C};\mathscr{C}_A))$ to be the multiset $|\mathscr{C}|_{\mathcal{Q}}$, ordered by the usual multiset extension of the ordering on the integers. Let $L(C)$ denote the number of Λ and λ-abstractions in C. Finally, let $P(C)$ denote the number of pairs that occur as arguments to flexible variables, let $UV(C)$ denote the sum, over each existentially quantified variable, of the number of variables universally quantified to the left of that variable in the quantifier context, let $EV(C)$ denote the number of existentially quantified variables which are bound in local quantifier contexts, and let $TV(C)$ denote the sum, over each universally quantified type variable, of the number of term variables universally quantified to the left of that variable in the quantifier context. We use the lexicographic ordering of $N(C)=\langle|V(\mathscr{L}(C))|,TS(C),L(C),P(C)+UV(C)+EV(C)+TV(C)\rangle$ to verify the termination of the algorithm.

The **Raise** transitions reduce $UV(C)$ while leaving the other components of the measure unchanged; since all of the applications introduced have flexible heads, the size measure $TS(C)$ is unchanged. The **RaiseTy** and **RaiseTm** transitions reduce $EV(C)$, and the **Permute** transitions $TV(C)$, while leaving the other measures unchanged. The **Lam-Lam** transitions reduce the total number of Λ and λ-abstractions $L(C)$ (while also possibly reduced the constraint size multiset $TS(C)$. The **Lam-Atom** rules may increase the $TS(C)$ measure (if the head of the atom is rigid), however they also reduce $L(C)$, so the measure as a whole is reduced. The **CurryTm** transition reduces the $P(C)$ measure. The **Rigid-Rigid** and **Pair-Pair** transitions reduce $TS(C)$ by replacing a constraint with a multiset of smaller constraints, while leaving the other measures

unchanged. The **Lam-Atom** rules reduce the number of λ-abstractions but increase the size of an application. If the head of the atom is flexible, the size of the application is zero, so the size multiset is unchanged, while the number of λ-abstractions is reduced. If the body of the λ-abstraction is rigid or a pair, any increase in the size of the application is offset by the removal of the λ-abstraction, so the size multiset is unchanged while, again, the number of λ-abstractions is reduced. If the head of the atom is rigid and the body of the λ-abstraction is a flexible atom, then in a finite number of **Lam-Atom** steps the constraint is simplified to constraints with one side flexible and the other rigid, and (as explained below) an application of a **Loc-Elim**, **Flex-Const** or **Flex-Param** rule reduces the vertex set measure.

For the case of the **Pair-Atom** rule, we have the transition:

$$(M_1, M_2) = L\,x[\overline{A_m}]\,\overline{N_n} \;\Rightarrow\; M_1 = (\mathbf{1}\,L)x[\overline{A_m}]\overline{N_n}, \qquad M_2 = (\mathbf{2}\,L)x[\overline{A_m}]\,\overline{N_n}.$$

The vertex set is clearly unchanged, since the head of the atom must be rigid in the application of the **Pair-Atom** rule. Furthermore the two constraints replacing the original constraint are smaller in size, so the multiset of constraint sizes is reduced by this rule. Note that we must use the multiset of constraint sizes here since the atom is duplicated by the rule in both of the new constraints.

For the case of **Loc-Elim**, we use the effect on the size of the locator tree to reason about termination. Let $M = A\bar{t}:\overline{K}\cdot\lambda\bar{x}:\overline{A}\cdot(G_1\,[\bar{t}]\bar{x}, G_2[\bar{t}]\bar{x})$ be the term which replaces F in the **Loc-Elim** rule. The first-order reduction of this is (G_1, G_2). Define $\bar{n}:[2]\to[2]$ by $\bar{1}=2$, $\bar{2}=1$. The locator tree for the configuration before the transition has a vertex labelled $(L\,i)F$, so by construction it must also have vertices labelled F, $i\,F$ and $\bar{i}\,F$. F is the root of a binary tree with child vertices $\mathbf{1}\,F$ and $\mathbf{2}\,F$. The transition for **Loc-Elim** replaces this tree with the two subtrees rooted at $\mathbf{1}\,F$ and $\mathbf{2}\,F$ (suitably relabelled to have roots G_1 and G_2). The resulting set of locator trees has its vertex count reduced by one, verifying that the first component of the termination measure is reduced by the **Loc-Elim** rule.

We are finally left with verifying the termination of the rules for eliminating flexible variables: **Flex-Pair**, **Flex-Const**, **Flex-Param**, **Flex-Flex-Same** and **Flex-Flex-Diff**. We only consider the case of **Flex-Pair**, since the other cases are straightforward:

- Claim 18, verified in the next subsection, considers the special case of termination for a list of constraints resulting from a flexible–rigid rule with an occurs check (**Flex-Const** or **Flex-Param**). One of cases that is considered is that of **Flex-Pair** (termination is simplified by the fact that the occurs check has been performed before **Flex-Pair** is applied). It is verified that the reduction of the subconstraint list terminates, and that when this reduction is complete the overall termination measure is increased.
- Claim 21, verified in Section 5.4, uses this claim to show that the overall termination measure is eventually reduced after an application of the **Flex-Pair** rule and the further reduction of the resulting subconstraints:

$$\mathscr{C}_1, \mathscr{C}_2, \ldots, \mathscr{C}_m \;\Rightarrow\; \mathscr{C}'_1, \ldots, \mathscr{C}'_k, \mathscr{C}_2, \ldots, \mathscr{C}_m \stackrel{*}{\Longrightarrow} \mathscr{C}_2, \ldots, \mathscr{C}_m$$

where the first transition is an application of **Flex-Pair** to $\mathscr{C}_1$. The further applications of these rules to $\mathscr{C}'_1,\ldots,\mathscr{C}'_k$ may lead, via rigid–rigid rules, to applications of the **Loc-Elim** and **Lam-Atom** rules.

We have verified that if configuration C' results from C as a result of applying a rule of the unification algorithm, then $N(C')<N(C)$ under the lexicographic ordering of such tuples. Since this measure is well-founded, the repeated application of the unification rules must eventually terminate, either with the empty constraint list or with a constraint list to which no further rules are applicable. □

5.3. *Solving a defining constraint list*

We verify that the repeated application of the algorithm to the list of subconstraints resulting from an application of **Flex-Const** or **Flex-Param** (with an occurs check) is guaranteed to terminate, and that the overall termination measure is reduced in the process. A *defining constraint list* is one of the form $(\mathscr{Q}^t_i\cdot(T_i\,\overline{t^i})=A_i)_{i\in[m]},(\mathscr{Q}^x_j(F_j[\overline{u^j}]\overline{x^j})=M_j)_{j\in[n]}$ (modulo reordering of term and type constraints) where:

(i) each T_i and F_j is existentially quantified and unique ($j_1\neq j_2$ implies $F_{j_1}\neq F_{j_2}$, and so on), and

(ii) for each $k\in[m]$, $T_k\notin(\bigcup_{i=1}^m FV(A_i))\cup(\bigcup_{j=1}^n FV(M_j))$, and for each $k\in[n]$, $F_k\notin\bigcup_{j=1}^n FV(M_j)$.

Claim 18. *The repeated application of the unification algorithm to a list of defining constraints is guaranteed to terminate, and moreover is guaranteed to reduce the overall termination measure.*

Proof. Given a quantifier context $\mathcal{Q}$, define the measure of a list of defining constraints $\mathscr{C}$ to be the following triple:

$$\langle(\text{Size of vertex set } |V(\mathscr{L}(\mathcal{Q};\mathscr{C}))|) - (\text{Length of constraint list } \mathscr{C}),$$
$$\{1+|M|_Q+|N|_Q|(M=N)\in\mathscr{R}[\![\mathscr{C}]\!](\mathcal{Q})\}\cup\{1+|A|_Q+|B|_Q|(A=B)\in\mathscr{R}[\![\mathscr{C}]\!](\mathcal{Q})\},$$
$$\text{Number of } \lambda \text{ and } \Lambda\text{-abstractions in } \mathscr{C}\rangle$$

where the second component of the triple is understood to be a multiset. An application of **Flex-Pair**, **Flex-Const** or **Flex-Param** to a defining constraint list $(F=M,\mathscr{C})$ replaces the first constraint with zero or more constraints while maintaining the properties of a defining constraint list. Since $F\notin FV(M)$, each of these replacement constraints is smaller in size than the original constraint. Furthermore the other constraints in $\mathscr{C}$ are unaffected by the resulting substitution, since the defined variable F does not occur in any of these constraints. Thus the multiset component of the triple is reduced by the application of one of these rules. Although the size of the vertex set is increased by the introduction of temporary variables $T_1,\ldots,T_{n_1},G_1,\ldots,G_{n_2}$ equated to the immediate subterms of M, each such new variable also has a defining constraint added to the front of the list, so the increase in the vertex set size is offset by the increase in

the constraint list length. Since the first component of the triple is unchanged and the second component reduced, the overall measure is reduced.

An application of **Flex-Flex-Diff** deletes a constraint without affecting the other constraints. By unifying two different variables, the vertex set size is reduced by one; since the constraint list length is reduced in the process, the first component of the measure is unchanged. Since the multiset component is reduced by the deletion of a constraint, the measure as a whole is reduced. If the **Flex-Flex-Same** rule were applied, the first component of the measure would be increased by the reduction in constraint list length; however the definition of a defining constraint list ensures that this cannot happen.

An application of **Loc-Elim** replaces a variable with a pair of variables. Although this may increase the term size multiset (if there are other occurrences of the replaced variable without locator applications), by an argument similar to before we have that the size of the vertex set is reduced, so the first component of the measure is reduced, and the measure as a whole reduced by the lexicographic ordering. Finally an application of **Lam-Atom** reduces the third component of the measure (the number of λ-abstractions) without affecting the term size multiset, since the application which is extended has a flexible head.

So the repeated applications of these rules to a list of defining constraints is guaranteed to terminate, and since one of these rules is always applicable to a defining constraint, eventually the list is reduced to the empty constraint list $\top$.

We conclude by considering the effect of these transitions on the underlying locator tree for the complete constraint list. We verify by induction on the number of transitions that the number of vertices in the locator tree is reduced by the elimination of the list of defining constraints. For the case of the **Flex-Const** and **Flex-Param** rules, the vertex for F is removed, while new vertices for $T_1,\ldots,T_{n_1}$, $G_1,\ldots,G_{n_2}$ are added to the tree, increasing the vertex count by n_1+n_2. Invoking the induction hypothesis on each of the defining constraints added for each of these new variables gives that the elimination of each of these n_1+n_2 constraints reduces the number of vertices of the locator trees, with a total reduction of least n_1+n_2 vertices in the locator tree, so the net effect of the elimination of the original constraint is to reduce the vertex count for the tree. A similar argument shows that the application of the **Flex-Pair** rule also reduces the number of vertices. An application of **Flex-Flex-Diff** merges two root vertices, and therefore their subtrees, again reducing the vertex count. Since none of the defined variables occur in the right hand sides, the **Flex-Flex-Same** rule is never applicable to a list of defining constraints. The **Loc-Elim** rule replaces a variable naming a vertex by two variables naming its two immediate sub-vertices, again reducing the vertex count. Finally an application of **Lam-Atom** leaves the vertex count and constraint size multiset unchanged, while reducing the total number of λ-abstractions, so the overall termination measure is reduced. □

When it is applicable, the **Flex-Flex-Same** rule leaves the vertex count unchanged while deleting a constraint, thus reducing the term size multiset measure and therefore the overall termination measure.

5.4. *Applying* **Flex-Pair** *and solving the result*

It remains to consider the general case of the application of the **Flex-Pair** rule without the occurs check.

Definition 19 (*Merged locator tree*). Define a *multi-equation* S to be a set of equated terms $\{M_i\}_i$ for $n \geqslant 1$. Syntactically multi-equations are defined by

$$S ::= \{\ \} \mid M \mid S_1 = S_2$$

where order is unimportant. We identify M with $M = \{\ \}$, and we identify $M = N$ with $M = N = \{\ \}$. A *merged vertex set* consists of multi-equations satisfying the following properties:

Prefix closure: If $(i\,L)F = S$ is a vertex (for some $i \in [2]$) then so is $LF = S'$ for some S'. Furthermore if $L'G$ is a component of S', then $(i\,L')G$ is a component of S.

Binary completeness: If $(\mathbf{1}\,L)F = S$ is a vertex, then so is $(\mathbf{2}\,L)F = S'$ for some S', and vice versa.

Given such a vertex set $\mathscr{V}$, define the *merged locator tree* $\mathscr{MT}(\mathscr{V})$ to consist of:

(i) vertices $V(\mathscr{MT}(\mathscr{V})) = \mathscr{V}$, and

(ii) edges $E(\mathscr{MT}(\mathscr{V})) = \{((LF = S), ((i\,L)F = S')) \mid (LF = S), ((i\,L)F = S') \in \mathscr{V}\}$

A *self-referring defining constraint list* (SRDCL) is a constraint list of the form $(\mathcal{Q}^j P_j = M_j)_{j \in [n]}$, where P is defined by:

$$P ::= \Lambda \overline{t} : \overline{K} \cdot \lambda \overline{x} : \overline{A} \cdot (P_1, P_2) \mid \Lambda \overline{t} : \overline{K} \cdot \lambda \overline{x} : \overline{A} \cdot F[\overline{p^t}]\overline{p^v}$$

where each variable in $\bigcup_j FV(P_j)$ is existentially quantified and has a unique occurrence in the set of left-hand sides $\{P_j\}_j$. Note that there may be repeated occurrences in the right-hand sides $\{M_j\}_j$, however this linearity restriction ensures that each variable has a single defining constraint. The linearity restriction also ensures that a SRDCL is closed under the application of any of the applicable unification rules (a variable in a left-hand side cannot be instantiated with a rigid atom). The termination construction below constructs a set of multi-equations from a SRDCL; this multi-equation set characterizes an acyclic directed graph obtained from a locator tree by identifying some vertices.

Definition 20 (*Merged locator tree construction*). Given a quantifier context $\mathcal{Q}$ and SRDCL $\mathscr{C}$, let $\mathscr{C}'$ be $\mathscr{R}[\![\mathscr{C}]\!](\mathcal{Q})$. For the purposes of this construction, consider a constraint to be an ordered pair rather than a set. Let $\mathscr{C}''$ be obtained from $\mathscr{C}'$ by exhaustively applying the following simplification rules:

$$\mathscr{C} \cup \{(M_1, N_1) = (M_2, N_2)\} \Rightarrow \mathscr{C} \cup \{M_1 = M_2,\ N_1 = N_2\}$$

$$\mathscr{C} \cup \{(M, N) = \mathsf{c}[\overline{A}]\overline{M}\} \Rightarrow \mathscr{C} \cup \{M = \mathsf{c}[\overline{A}]\overline{M}, N = \mathsf{c}[\overline{A}]\overline{M}\}$$

$$\mathscr{C} \cup \{LF = (M, N)\} \Rightarrow \mathscr{C} \cup \{(\mathbf{1}\,L)F = M, (\mathbf{2}\,L)F = N\}$$

$$\mathscr{C} \cup \{(M, N) = LF\} \Rightarrow \mathscr{C} \cup \{M = (\mathbf{1}\,L)F, N = (\mathbf{2}\,L)F\}.$$

$\mathscr{C}''$ is a set of multi-equations of the form $LF = M$, where M is an atom (rigid or flexible). Let $\mathscr{V}$ be the union of $\{(LF = L'F') \in \mathscr{C}''\}$ and $\{L_j F_j\}_j$, the flexible atoms occurring in $\mathscr{C}''$. $\mathscr{V}$ is a set of multi-equations; let $\mathscr{V}'$ result from exhaustively applying the following merging transitions to $\mathscr{V}$:

$$\mathscr{V} \cup \{(L_F F = L_G G = S_1), ((i L_F) F = S_2)\}$$
$$\Rightarrow \mathscr{V} \cup \{(L_F F = L_G G = S_1), ((i L_G)\ G = (i L_F) F = S_2)\}$$
where $((i L_G) G)$ not in S_2

$$\mathscr{V} \cup \{(LF = S_1), (LF = S_2)\} \Rightarrow \mathscr{V} \cup \{(LF = S_1 = S_2)\}$$

$$\mathscr{V} \cup \{(i L) F = S\} \Rightarrow \mathscr{V} \cup \{((i L) F = S), LF\}$$
where (LF) is not a component in $\mathscr{V}$

$$\mathscr{V} \cup \{(\mathbf{1} L) F = S\} \Rightarrow \mathscr{V} \cup \{((\mathbf{1} L) F = S), (\mathbf{2} L) F\}$$
where $((\mathbf{2} L) F)$ is not a component in $\mathscr{V}$

$$\mathscr{V} \cup \{(\mathbf{2} L) F = S\} \Rightarrow \mathscr{V} \cup \{((\mathbf{2} L) F = S), (\mathbf{1} L) F\}$$
where $((\mathbf{1} L) F)$ is not a component in $\mathscr{V}$.

This rewriting process can be verified to terminate by exhaustively applying the first two merging rules, then exhaustively applying the latter three completion rules. Finally define the merged locator tree $\mathscr{MG}(\mathcal{Q}; \mathscr{C})$ to be $\mathscr{MT}(\mathscr{V}')$.

Given a quantifier context $\mathcal{Q}$ and SRDCL $\mathscr{C}$, let $\{L_1^i F_1^i = \cdots = L_{n_i}^i F_{n_i}^i\}_i$ be $V(\mathscr{MG}(\mathcal{Q}; \mathscr{C}))$. Define the measure of $\mathscr{C}$ to be the triple:

$$\left\langle \sum_i n_i,\ |\mathscr{C}|_{\mathcal{Q}},\ \text{number of } \lambda\text{-abstractions in } \mathscr{C} \right\rangle.$$

Claim 21. *The repeated application of the unification algorithm to a SRDCL is guaranteed to terminate, and moreover is guaranteed to reduce the overall termination measure.*

Proof. We verify that the application of a unification transition to a SRDCL reduces this measure. The **Lam-Lam**, **Pair-Pair** and **Pair-Atom** rules obviously reduce the (second or third) component of the measure. The problematic case is an application of **Lam-Atom** where the λ-abstraction is flexible and the atom rigid. After a finite number of applications of **Lam-Atom** rules, the constraint is of the form $\mathcal{Q} \cdot F[\overline{p^l}]\overline{p^v} = t[\overline{A}]\overline{M}$. If the defined variable (T or F) has a free occurrence in the other hand side of the constraint, then no further unification transitions are applicable; the constraints are not satisfiable. If theree is no such free occurrence, a **Flex-Const** or **Flex-Param** rule is applicable. In this case the single constraint to which the rule is applicable is a defining constraint list of length one; the argument before verifies that this constraint list is eventually reduced to the empty constraint list, while reducing the overall locator tree measure. It is left to verify that the merged locator tree size measure is also reduced.

The substitution for the defined variable (T or F) removes an element of this set or reduces the size of an element of this set by one; the remaining case to consider is whether an application of a **Loc-Elim** rule can introduce an outermost pair constructor in the remaining constraints in the SRDCL which increases the merged locator tree measure. Consider for example:

Substitution	Constraint list	Multi-equation set
-	$F = \mathsf{C}(\mathbf{1}\,G), G = H$	$F, G = H, \mathbf{1}\,G = \mathbf{1}\,H, \mathbf{2}\,G = \mathbf{2}\,H$
$\{(\mathsf{C}\,F')/F\}$	$F' = \mathbf{1}\,G, G = H$	$G = H, F' = \mathbf{1}\,G = \mathbf{1}\,H, \mathbf{2}\,G = \mathbf{2}\,H$
$\{(G_1, G_2)/G\}$	$F' = G_1, (G_1, G_2) = H$	$F' = G_1, \mathbf{1}\,H = G_1, \mathbf{2}\,H = G_2$
$\{G_1/F'\}$	$(G_1, G_2) = H$	$G_1 = \mathbf{1}\,H, G_2 = \mathbf{2}\,H$

However (as in this example) any variable F which is pair-expanded by **Loc-Elim** during the elimination of a flexible–rigid constraint must therefore have a located occurrence $(L\,i)F$, so the merged locator tree construction includes the addition of elements for $(L\,1)F$ and $(L\,2)F$. If there is a constraint of the form $F = L'G$, then the merged locator tree construction merges the elements for $L''F$ and $(L''L')\,G$, for each prefix L'' of $L\,i$, in the process deepening the locator tree measure for G. The **Loc-Elim** measure corresponds to naming the two subvertices of F in the locator tree rooted at F, in the process removing any reference to F. So the merged locator tree measure is reduced by the application for **Flex-Const** or **Flex-Param**, so the application of **Lam-Atom** leads in a finite number of steps to the reduction of the first component of the measure.

We have also verified that applications of the **Flex-Const**, **Flex-Param** and **Loc-Elim** rules reduce the merged locator tree measure. Similarly an application of **Flex-Flex-Diff** reduces the merged locator tree measure; an application of **Flex-Flex-Same** leaves this measure unchanged while reducing the constraint size multiset. Finally the reduction in the merged locator tree measure caused by the application of **Flex-Pair** again follows a reasoning similar to that for the **Loc-Elim** rule (again using the fact that because there are no non-rigid recursive self-references, the underlying locator graph is acyclic and therefore the depth of the merged locator tree rooted at a particular variable is guaranteed to be finite).

It is finally left to verify that the overall termination measure is reduced by the elimination of a flexible-pair constraint. We verify this by induction on the unification transitions which eliminate this constraint. The case for most of the rules is straightforward: for the **Flex-Const** and **Flex-Param** rules, the locator tree measure is reduced as shown in the proof for Claim 18, while the case for **Loc-Elim** is as usual. We consider then the case for **Flex-Pair**, applied to a constraint of the form $\mathcal{Q} \cdot F[\overline{p^l}]\overline{p^v} = (M, N)$. If there is a located occurrence of F anywhere in the constraint list, then the locator tree measure is reduced by the application of **Flex-Pair** in the same way that it is reduced by **Loc-Elim**. The **Flex-Pair** rule introduces the subconstraints $\mathcal{Q} \cdot H_1[\overline{p^l}]\overline{p^v} = M$ and $\mathcal{Q} \cdot H_2[\overline{p^l}]\overline{p^v} = N$ for some new H_1, H_2. The exhaustive application of the rules to these constraints reduces the locator tree measure further or leaves it unchanged (the latter may happen because of applications of the **Flex-Flex-Same** rules; consider for

example $F = (\mathbf{1}F, \mathbf{2}F)$, which is reduced by **Flex-Pair** to $H_1 = H_1, H_2 = H_2$). So eventually all subconstraints are eliminated with the locator tree measure strictly reduced by the elimination of F. If on the other hand there are no located occurrences of F, then we replace the vertex for F with vertices for H_1 and H_2, with new defining constraints for H_1 and H_2. Because of the absence of the occurs check for **Flex-Pair** rules, each of these subconstraints may contain rigid recursive self-references if the terms are not unifiable; however in the absence of located occurrences of F, there cannot be any non-rigid self-references in the original constraint, so the **Flex-Flex-Same** rule is never applicable to subconstraints. If the terms are unifiable, then invoking the induction hypothesis gives that these constraints are each eliminated while strictly reducing the locator tree measure, so the net effect of eliminating this flexible-pair constraint is to reduce the locator tree measure. □

6. Correctness of the algorithm

We use Miller's unification logic, as formulated by Pfenning [16, 23], to reason about correctness of the unification algorithm, and also to reason about unifying substitutions. We use the judgement form $\Gamma \rhd_\Sigma \mathscr{F}$ to denote derivability in the unification logic of the formula $\mathscr{F}$. We also wish to reason about the well-typedness of substitutions in the context of the raising rules. We therefore introduce the new judgement form $\Gamma \rhd_\Sigma \sigma : \mathscr{F}$ to reason about well-typedness of the substitution σ with respect to the unification formula $\mathscr{F}$.

Definition 22 (*Unification logic*). Terms of the *extended unification logic* [16, 25] are defined to be the formulae of the judgement forms defined in Section A, augmented with the following:

$$\mathscr{F} ::= \forall x : A \cdot \mathscr{F} \mid \forall t : K \cdot \mathscr{F} \mid \exists x : A \cdot \mathscr{F} \mid \exists t : K \cdot \mathscr{F} \mid \mathscr{F}_1, \mathscr{F}_2 \mid \top.$$

Derivability in the unification logic is given by the derivation rules given in Section A augmented with the following:

$$\frac{\Gamma \rhd_\Sigma A \in K \quad \Gamma \rhd_\Sigma \{A/t\}\mathscr{F}}{\Gamma \rhd_\Sigma \exists t : K \cdot \mathscr{F}} \qquad \frac{\Gamma, t : K \rhd_\Sigma \mathscr{F}}{\Gamma \rhd_\Sigma \forall t : K \cdot \mathscr{F}}$$

$$\frac{\Gamma \rhd_\Sigma M \in A \quad \Gamma \rhd_\Sigma \{M/x\}\mathscr{F}}{\Gamma \rhd_\Sigma \exists x : A \cdot \mathscr{F}} \qquad \frac{\Gamma, x : A \rhd_\Sigma \mathscr{F}}{\Gamma \rhd_\Sigma \forall x : A \cdot \mathscr{F}}$$

$$\frac{\Gamma \rhd_\Sigma \mathscr{F}_1 \quad \Gamma \rhd_\Sigma \mathscr{F}_2}{\Gamma \rhd_\Sigma \mathscr{F}_1, \mathscr{F}_2} \qquad \frac{\Gamma \mathbf{env}_\Sigma}{\Gamma \rhd_\Sigma \top}.$$

We will assume without loss of generality that all variables quantified in a formula of the unification logic are distinct. This can be ensured by formulae representing configurations of the algorithm if bound variables are renamed appropriately before terms are analysed by the algorithm.

A configuration $(\mathcal{Q};\mathscr{C};\mathscr{C}_A)$ can be considered as a formula of the unification logic $\mathcal{Q}(\mathscr{C},\mathscr{C}_A)$. A valid *initial configuration* is one of the form $(\mathcal{Q};\mathscr{C};\top)$ where, for $\mathcal{Q}'\cdot M = N$ in $\mathscr{C}$, we have $\Vdash_\Sigma \mathcal{Q}\mathcal{Q}'\cdot M\in A$ and $\Vdash_\Sigma \mathcal{Q}\mathcal{Q}'\cdot N\in A$ for some type A, and for $\mathcal{Q}'\cdot A = B$ in $\mathscr{C}$, we have $\Vdash_\Sigma \mathcal{Q}\mathcal{Q}'\cdot A\in K$ and $\Vdash_\Sigma \mathcal{Q}\mathcal{Q}'\cdot B\in K$ for some kind K, where $\Gamma \Vdash_\Sigma \mathscr{F}$ denotes derivability in the unification logic presented above.

A potential problem arises with the fact the inference rule for the existential includes a side premise verifying that the witness term is well-typed. The well-typedness of terms may in some cases depend on the unifiability of some other terms. This is true even if the terms being unified have already been verified to have a common type. Consider for example the equality constraint [5]:

$$\mathsf{c}\,[A]\,M = \mathsf{c}\,[B]\,N$$

where c is a constant with type $\Delta t:\mathsf{Type}\cdot t\to \mathsf{int}$. In this case the well-typedness of any substitutions arising from unifying M and N depends on the unifiability of A and B. We avoid this problem by ensuring that the unification algorithm only constructs well-typed substitutions. This ensures that terms constructed by the algorithm are well-typed.

Lemma 23. *Given an initial configuration* $(\mathcal{Q};\mathscr{C};\mathscr{C}_A)$, *if* $(\mathcal{Q};\mathscr{C};\mathscr{C}_A)\stackrel{*}{\Rightarrow}(\mathcal{Q}';\mathscr{C}';\mathscr{C}'_A)$, *then:*

(i) *For each constraint* $x = M$ *in* $\mathscr{C}'_A$, *the types of* M *and* x *are equal. For each constraint* $t = A$ *in* $\mathscr{C}'_A$, *the kinds of* A *and* t *are equal.*

(ii) *For any constraint* $\mathcal{Q}\cdot M = N$ *which is analysed during this execution,* M *and* N *have equal types. Similarly for any constraint* $\mathcal{Q}\cdot A = B$.

To state a completeness result for the interpreter we need to reason about the possible substitutions for the "free" (existentially quantified) variables in a configuration.

Definition 24 (*$\mathscr{F}$-substitution*). A *substitution* σ is a finite set of ordered (variable, term) pairs, $\{(t_i,A_i)\}_i\cup\{(x_j,M_j)\}_j$, where $t_{i_1}=t_{i_2}$ implies $i_1=i_2$; and $x_{j_1}=x_{j_2}$ implies $j_1=j_2$. σ is a (*closed*) *$\mathscr{F}$-substitution* if $\Vdash_\Sigma \sigma:\mathscr{F}$, where derivability is defined by the following rules:

$$\frac{t\in dom(\sigma)\quad \Gamma\rhd_\Sigma \sigma(t)\in K\quad \Gamma\rhd_\Sigma \sigma:\{\sigma(t)/t\}\mathscr{F}}{\Gamma\rhd_\Sigma \sigma:\exists t:K\cdot\mathscr{F}}\qquad \frac{t\notin dom(\sigma)\quad \Gamma,t:K\rhd_\Sigma \sigma:\mathscr{F}}{\Gamma\rhd_\Sigma \sigma:\forall t:K\cdot\mathscr{F}}$$

$$\frac{x\in dom(\sigma)\quad \Gamma\rhd_\Sigma \sigma(x)\in A\quad \Gamma\rhd_\Sigma \sigma:\{\sigma(x)/x\}\mathscr{F}}{\Gamma\rhd_\Sigma \sigma:\exists x:A\cdot\mathscr{F}}\qquad \frac{x\notin dom(\sigma)\quad \Gamma,x:A\rhd_\Sigma \sigma:\mathscr{F}}{\Gamma\rhd_\Sigma \sigma:\forall x:A\cdot\mathscr{F}}$$

$$\frac{\Gamma\rhd_\Sigma \sigma:\mathscr{F}_1\quad \Gamma\rhd_\Sigma \sigma:\mathscr{F}_2}{\Gamma\rhd_\Sigma \sigma:\mathscr{F}_1,\mathscr{F}_2}\qquad \frac{\mathscr{F}\ \textit{is atomic}\quad \Gamma\rhd_\Sigma \mathscr{F}}{\Gamma\rhd_\Sigma \sigma:\mathscr{F}}.$$

Note that the last judgement form includes type and kind membership judgements and equality judgements. We will denote derivability of the unification logic judgement $\Gamma\rhd_\Sigma\mathscr{F}$ according the rules in Definition 22 by $\Gamma\Vdash_\Sigma\mathscr{F}$. We will denote derivability of the judgement $\Gamma\rhd_\Sigma\sigma:\mathscr{F}$ according the rules in Definition 24 by $\Gamma\Vdash_\Sigma\sigma:\mathscr{F}$.

A final notion that will be useful for reasoning about the correctness of the raising transitions is the following:

Definition 25. For substitutions σ and σ', we say that $\sigma \sqsubseteq \sigma'$ if $dom(\sigma) \subseteq dom(\sigma')$ and there exist σ_1 and σ_2 with $dom(\sigma) = dom(\sigma_1) = dom(\sigma_2)$ and, for all $x \in dom(\sigma)$, $\sigma(x) = \sigma'(\sigma_1(x))$ and $\sigma'(x) = \sigma(\sigma_2(x))$.

Theorem 26. *Assume given an interpreter configuration $(\mathcal{Q};\mathscr{C};\mathscr{C}_A)$ such that each term in $\mathscr{C}$ satisfies the pattern restriction. If $(\mathcal{Q};\mathscr{C};\mathscr{C}_A) \stackrel{*}{\Rightarrow} (\mathcal{Q}';\mathscr{C}';\mathscr{C}'_A)$, then*

(i) *$\Vdash_\Sigma \mathcal{Q}(\mathscr{C},\mathscr{C}_A)$ if and only if $\Vdash_\Sigma \mathcal{Q}'(\mathscr{C}',\mathscr{C}'_A)$.*

(ii) *If $\Vdash_\Sigma \sigma : \mathcal{Q}(\mathscr{C},\mathscr{C}_A)$ then there exists some σ' such that $\sigma \sqsubseteq \sigma'$ and $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}', \mathscr{C}'_A)$.*

(iii) *If $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}',\mathscr{C}'_A)$ then there exists some σ such that $\sigma \sqsubseteq \sigma'$ and $\Vdash_\Sigma \sigma : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$.*

(iv) *For any constraint $t = A$ added to $\mathscr{C}'_A$, the kinds of t and A are equal. For any constraint $x = M$ added to $\mathscr{C}'_A$, the types of x and M are equal according to the equality theory for the type system.*

(v) *For any constraint $\mathcal{Q}(A = B)$ which is analysed by the algorithm, A and B have equal kinds. For any constraint $\mathcal{Q}(M = N)$ which is analysed by the algorithm, M and N have equal types according to the equality theory for the type system.*

The substitution properties for the unification algorithm rely on the fact that the unification algorithm does not remove an existential quantifier once it has been introduced. On the other hand the only occurrence of an existentially quantified variable t or x may be on the left hand side of a solved constraint $t = A$ and $x = M$. In some steps of the proof, we use `locator`$(L\ x)$ to denote L.

Proof. We verify the theorem by induction on the length of the transition sequence. (1) follows from (2) and (3). (4) and (5) are verified by a straightforward induction on the application of the transition rules, using the correctness according to (2) and (3) and the fact that the constraint list $\mathscr{C}$ is processed in LIFO order. We concentrate therefore on verifying (2) and (3).

The cases for the rigid–rigid rules are fairly straightforward. The completeness of the **Lam-Atom** rules are justified by an application of ΔI or $\rightarrow I$ followed by an application of η-reduction. The soundness of the **Pair-Atom** rules are justified by an application of $\times I$ followed by an application of η-reduction of the right-hand side. Soundness of the rules follows by the Church–Rosser property for the original calculus.

The raising rules are the only case where we need the general definition of $\sigma \sqsubseteq \sigma$. Consider for example the **Raise**$_{KT}$ rule, and suppose $\Vdash_\Sigma \sigma : \mathcal{Q}(\mathscr{C},\mathscr{C}_A)$. Define $\sigma' = \sigma - \{(F,\sigma(F))\} \cup \{(F, \Lambda t : K \cdot \sigma(F))\}, \sigma_1(F) = F[t], \sigma_2(F) = \Lambda t : K \cdot F$, then (2) follows fairly obviously. On the other hand, if $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}',\mathscr{C}'_A)$, then let $\sigma = \sigma' - \{(F,\sigma'(F))\} \cup \{(F,\sigma'(F)[t])\}$, with σ_1 and σ_2 as before.

For the parts of the algorithm not using the raising transitions, it suffices to consider $\sigma \subseteq \sigma'$ rather than $\sigma \sqsubseteq \sigma'$, taking σ_1 and σ_2 as the identity substitutions. For Part (3) of the theorem, we take $\sigma = \sigma'$.

For the soundness of the **Loc-Elim** rule, assume $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}', \mathscr{C}'_A)$, so $\sigma'(F) = \sigma'(N)$ where $N = \Lambda \vec{t} : \overline{K} . \lambda \vec{x} : \overline{A} . (G_1[\vec{t}]\vec{x}, G_2[\vec{t}]\vec{x})$. By the definition of the algorithm, existentially quantified variables in $\mathcal{Q}$ are retained in $\mathcal{Q}'$ so σ' is defined for terms with flexible variables bound in $\mathcal{Q}$. Then for any subterm M in $\mathscr{C}, \mathscr{C}_A$:

$$\begin{aligned}\sigma'(\{N/F\}M) &= (\sigma' \cup \{\sigma'(N)/F\})(M) \\ &= \sigma'(M).\end{aligned}$$

So $\Vdash_\Sigma \sigma' : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$. For completeness, assume $\Vdash_\Sigma \sigma : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$ and let $\sigma' = \{(\Lambda \vec{t} : \overline{K} . \lambda \vec{x} : \overline{\sigma(A)} . \mathbf{1}(\sigma(F)[\vec{t}]\vec{x})/G_1, (\Lambda \vec{t} : \overline{K} . \lambda \vec{x} : \overline{\sigma(A)} . \mathbf{2}(\sigma(F)[\vec{t}]\vec{x})/G_2\} \cup \sigma$. Then:

$$\begin{aligned}\sigma'(F) &= \sigma'(\Lambda \vec{t} : \overline{K} . \lambda \vec{x} : \overline{A} . (\mathbf{1}(F[\vec{t}]\vec{x}), \mathbf{2}(G_2[\vec{t}]\vec{x}))) \\ &= \Lambda \vec{t} : \overline{K} . \lambda \vec{x} : \overline{\sigma'(A)} . (\mathbf{1}(\sigma'(F)[\vec{t}]\vec{x}), \mathbf{2}(\sigma'(F)[\vec{t}]\vec{x})) \\ &= \Lambda \vec{t} : \overline{K} . \lambda \vec{x} : \overline{\sigma(A)} . (\mathbf{1}(\sigma(F)[\vec{t}]\vec{x}), \mathbf{2}(\sigma(F)[\vec{t}]\vec{x})) \\ &= \Lambda \vec{t} : \overline{K} . \lambda \vec{x} : \overline{\sigma(A)} . (\sigma'(G_1)[\vec{t}]\vec{x}, \sigma'(G_2)[\vec{t}]\vec{x}) \\ &= \sigma'(\Lambda \vec{t} : \overline{K} . \lambda \vec{x} : \overline{A} . (G_1[\vec{t}]\vec{x}, G_2[\vec{t}]\vec{x})) \\ &= \sigma'(N).\end{aligned}$$

So $\sigma'(\{N/F\}M) = \sigma'(M)$ for any subterm M in $\mathscr{C}', \mathscr{C}'_A$, by a reasoning similar to before, so $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}', \mathscr{C}'_A)$.

For the currying transition **CurryTm**, suppose $\Vdash_\Sigma \sigma : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$ and let $\sigma' = \{(\Lambda \overline{t_k} : \overline{K_k} . \lambda \overline{y_m} : \overline{A_m} . \lambda y_{B_1} : (\overline{A_N} \to A_{N_1}) . \lambda y_{B_2} : (\overline{A_N} \to A_{N_2}) . \lambda \overline{y'_n} : \overline{A'_n} . \sigma(F)[\overline{t_k}]\overline{y_m}(y_{B_1}, y_{B_2})\overline{y'_n})/G\} \cup \sigma$. Then:

$$\begin{aligned}\sigma'(F) &= \sigma'(\Lambda \overline{t_k} : \overline{K_k} . \lambda \overline{y_m} : \overline{A_m} . \lambda y_B : (\overline{A_N} \to A_{N_1} \times A_{N_2}) . \lambda \overline{y'_n} : \overline{A'_n} . F[\overline{t_k}]\overline{y_m} y_B \overline{y'_n}) \\ &= \Lambda \overline{t_k} : \overline{K_k} . \lambda \overline{y_m} : \overline{A_m} . \lambda y_B : (\overline{A_N} \to A_{N_1} \times A_{N_2}) . \lambda \overline{y'_n} : \overline{A'_n} . \sigma'(F)[\overline{t_k}]\overline{y_m} y_B \overline{y'_n} \\ &= \Lambda \overline{t_k} : \overline{K_k} . \lambda \overline{y_m} : \overline{A_m} . \lambda y_B : (\overline{A_N} \to A_{N_1} \times A_{N_2}) . \lambda \overline{y'_n} : \overline{A'_n} . \sigma(F)[\overline{t_k}]\overline{y_m} y_B \overline{y'_n} \\ &= \Lambda \overline{t_k} : \overline{K_k} . \lambda \overline{y_m} : \overline{A_m} . \lambda y_B : (\overline{A_N} \to A_{N_1} \times A_{N_2}) . \lambda \overline{y'_n} : \overline{A'_n} . \\ &\quad \sigma(F)[\overline{t_k}]\overline{y_m}(\mathbf{1}\, y_B, \mathbf{2}\, y_B)\overline{y'_n} \\ &= \Lambda \overline{t_k} : \overline{K_k} . \lambda \overline{y_m} : \overline{A_m} . \lambda y_B : (\overline{A_N} \to A_{N_1} \times A_{N_2}) . \lambda \overline{y'_n} : \overline{A'_n} . \\ &\quad \sigma'(G)[\overline{t_k}]\overline{y_m}(\mathbf{1}\, y_B)(\mathbf{2}\, y_B)\overline{y'_n} \\ &= \sigma'(N).\end{aligned}$$

To verify the **Flex-Pair** rule, suppose $\Vdash_\Sigma \sigma : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$ and let $\sigma' = \{(\Lambda \overline{t_k} : \overline{K_k} . \lambda \overline{x_m} : \overline{A_m} . i(\sigma(F)[\overline{t_k}]\overline{x_m}))/G_i\}_{i \in [2]} \cup \sigma$. As above we have $\sigma'(F) = \sigma'(N)$ and furthermore:

$$\begin{aligned}\sigma'(G_i[\overline{p_k^t}]\overline{p_m^x}) &= (\Lambda \overline{t_k} : \overline{K_k} . \lambda \overline{x_m} : \overline{A_m} . i(\sigma(F)[\overline{t_k}]\overline{x_m}))[\overline{p_k^t}]\overline{p_m^x} \\ &= i(\sigma(F)[\overline{p_k^t}]\overline{p_m^x})\end{aligned}$$

$$= \sigma(i(F[\overline{p_k^t}]\overline{p_m^x}))$$
$$= \sigma(M_i).$$

In the other direction, assume $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}', \mathscr{C}'_A)$ then:

$$\sigma'(F[\overline{p_k^t}]\overline{p_m^x}) = \sigma'(\Lambda\overline{t_k} : \overline{K_k} \cdot \lambda\overline{x_m} : \overline{A_m} \cdot (G_1[\overline{t_k}]\overline{x_m}, G_2[\overline{t_k}]\overline{x_m}))[\overline{p_k^t}]\overline{p_m^x}$$
$$= (\sigma'(G_1)[\overline{t_k}]\overline{x_m}, \sigma'(G_2)[\overline{t_k}]\overline{x_m})$$
$$= (\sigma'(M_1), \sigma'(M_2)).$$

To verify the **Flex-Param** rule, assume $\Vdash_\Sigma \sigma : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$ and let $(\Lambda\overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda\overline{x_{m_2}} : \overline{A_{m_2}} \cdot L'$ $x_k[\overline{B'_{n'_1}}]\overline{M'_{n'_2}}) = \sigma(F)$, then $\sigma(F[\overline{p^t_{m_1}}]\overline{p^x_{m_2}}) = \{\overline{p^t_{m_1}}/\overline{t_{m_1}}, \overline{p^x_{m_2}}/\overline{x_{m_2}}\}(L'x_k[\overline{B'_{n'_1}}]\overline{M'_{n'_2}}) = \sigma(Lz[\overline{B_{n_1}}]$ $\overline{M_{n_2}})$. An inductive argument on $\max(n_1, n'_1)$ demonstrates that $n_1 = n'_1$ and $\{\overline{p^t_{m_1}}/\overline{t_{m_1}}\}B'_i$ $= \sigma(B_i)$ for each $i \in [n_1]$, and an inductive argument on $\max(n_2, n'_2)$ demonstrates that $n_2 = n'_2$ and $\{\overline{p^t_{m_1}}/\overline{t_{m_1}}, \overline{p^x_{m_2}}/\overline{x_{m_2}}\}M'_j = \sigma(M_j)$ for each $j \in [n_2]$. Furthermore $\{\overline{p^x_{m_2}}/\overline{x_{m_2}}\}L'x_k$ $= (Lz)$; since $z \notin FV(\sigma(F))$ we must have $z = x_k$ for some k and $\{\overline{p^x_{m_2}}/\overline{x_{m_2}}\}x_k = q^x_k$ such that $\mathtt{source}(q^x_k) = z$ and L' $\mathtt{locator}(q^x_k) = L$. By the extended pattern restriction there is a unique choice for such a k, so we have $(Lz) = (L'p^x_k)$. Define $\sigma' = \sigma \cup \{(\lambda\overline{t_{m_1}} : \overline{K_{m_1}} \cdot$ $B'_i)/T_i\}_{i\in[n_1]} \cup \{(\Lambda\overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda\overline{x_{m_2}} : \overline{A_{m_2}} \cdot M'_j)/G_j\}_{j\in[n_2]}$. Then we have

$$\sigma'(F) = \Lambda\overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda\overline{x_{m_2}} : \overline{A_{m_2}} \cdot L'x_k[\overline{B'_{n'_1}}]\overline{M'_{n'_2}}$$
$$= \Lambda\overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda\overline{x_{m_2}} : \overline{A_{m_2}} \cdot L'x_k[(\overline{T_n\overline{t_{m_1}}})](\overline{G_{n_2}[\overline{p^t_{m_1}}]\overline{p^x_{m_2}}})$$
$$= \sigma'(N),$$

where $N = \Lambda\overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda\overline{x_{m_2}} : \overline{A_{m_2}} \cdot L'x_k[(T_1\overline{p^t_{m_1}}) \ldots (T_{n_1}\overline{p^t_{m_1}})](G_1[\overline{p^t_{m_1}}]\overline{p^x_{m_2}}) \ldots (G_{n_2}[\overline{p^t_{m_1}}]$ $\overline{p^x_{m_2}})$. So $\sigma(M) = \sigma'(M) = (\sigma' \cup \{\sigma'(F)/F\})(M) = (\sigma' \cup \{\sigma'(N)/F\})(M) = \sigma'(\{N/F\}M)$ for any term M in $\mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$. Furthermore, since $L'x_k[\overline{\sigma'(B_{n_1})}]\overline{\sigma'(M_{n_2})} = \sigma'(F[\overline{p^t_{m_1}}]\overline{p^x_{m_2}})$ $= L'x_k[\overline{\{p^t_{m_1}/t_{m_1}\}B'_{n_1}}]\overline{\{p^t_{m_1}/t_{m_1}, p^x_{m_2}/x_{m_2}\}M'_{n_2}}$, we have $\sigma'(B_i) = \overline{\{p^t_{m_1}/t_{m_1}\}B'_i} = \sigma'(T_i)\overline{p^t_{m_1}}$ for $i \in [n_1]$, and $\sigma'(M_j) = \{\overline{p^t_{m_1}/t_{m_1}}, \overline{p^x_{m_2}/x_{m_2}}\}M'_j = \sigma'(G_j)[\overline{p^t_{m_1}}]\overline{p^x_{m_2}}$ for $j \in [m_2]$.

In the other direction assume $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}', \mathscr{C}'_A)$. Since $\sigma'(F) = \sigma'(N)$ we have as usual $\sigma'(\{N/F\}M) = \sigma'(M)$ for any term M in $\mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$. Furthermore, we have

$$\sigma'(F[\overline{p^t_{m_1}}]\overline{p^x_{m_2}})$$
$$= \sigma'(N)[\overline{p^t_{m_1}}]\overline{p^x_{m_2}}$$
$$= \{\overline{p^t_{m_1}/t_{m_1}}, \overline{p^x_{m_2}/x_{m_2}}\}(L'x_k[(\sigma'(T_1)\overline{t_{m_1}}) \ldots (\sigma'(T_{n_1})\overline{t_{m_1}})]$$
$$(\sigma'(G_1)[\overline{t_{m_1}}]\overline{x_{m_2}}) \ldots (\sigma'(G_{n_2})[\overline{p^t_{m_1}}]\overline{p^x_{m_2}}))$$
$$= (L'(L''z))[\sigma'(T_1\overline{t_{m_1}}) \ldots \sigma'(T_{n_1}\overline{t_{m_1}})]\sigma'(G_1[\overline{t_{m_1}}]\overline{x_{m_2}}) \ldots \sigma'(G_{n_2}[\overline{p^t_{m_1}}]\overline{p^x_{m_2}})$$
$$= Lz[\sigma'(B'_1) \ldots \sigma'(B'_{n_1})]\sigma'(M'_1) \ldots \sigma'(M'_{n_2}).$$

So we conclude $\Vdash_\Sigma \sigma' : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$.

The case for **Flex-Const** is similar. We consider finally the cases of **Flex-Flex-Same** and **Flex-Flex-Diff**. Consider first the case for **Flex-Flex-Same** and assume $\Vdash_\Sigma$

$\sigma : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$. Then we have $\sigma(F) = \Lambda\overline{t_m} : \overline{K_m} \cdot \lambda\overline{x_n} : \overline{A_n} \cdot M$ for some M and $\{\overline{p_m^t}/\overline{t_m}, \overline{p_n^x}/\overline{x_n}\}M = \sigma(F[\overline{p_m^t}]\overline{p_n^x}) = \sigma(F[\overline{q_m^t}]\overline{q_n^x}) = \{\overline{q_m^t}/\overline{t_m}, \overline{p_n^x}/\overline{x_n}\}M$. Thus if $p_k^t \neq q_k^t$ for any $k \in [m]$ we must have $t_k \notin FV(M)$; similarly, if $p_k^x \neq q_k^x$ for any $k \in [n]$ we must have $x_k \notin FV(M)$. Letting $I = \{i_j\}_{j \in [m']} \subseteq [m]$ and $K = \{k_j\}_{j \in [n']} \subseteq [n]$ be the index sets described by the transition rule, so $\{\{t_i\}_{i \in I} \cup \{x_k\}_{k \in K} \subseteq FV(M)\}$, let $\sigma' = \sigma \cup \{(\Lambda t_{i_1} : K_{i_1} \cdots \Lambda t_{i_{m'}} : K_{i_{m'}} \cdot \lambda x_{k_1} : A_{k_1} \cdots \lambda x_{k_{n'}} : A_{k_{n'}} \cdot M)/G\}$. Then $\sigma'(N) = \sigma'(\Lambda\overline{t_m} : \overline{K_m} \cdot \lambda\overline{x_n} : \overline{A_n} \cdot G[t_{i_1} \ldots t_{i_{m'}}] x_{k_1} \ldots x_{k_{n'}}) = \Lambda\overline{t_m} : \overline{K_m} \cdot \lambda\overline{x_n} : \overline{A_n} \cdot \sigma'(G)[t_{i_1} \ldots t_{i_{m'}}]x_{k_1} \ldots x_{k_{n'}} = \Lambda\overline{t_m} : \overline{K_m} \cdot \lambda\overline{x_n} : \overline{A_n} \cdot M = \sigma'(F)$. And for any term M' in $\mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$ we have $\sigma'(M') = (\sigma' \cup \{\sigma'(F)/F\})(M') = (\sigma' \cup \{\sigma'(N)/F\})(M') = \sigma'(\{N/F\}M')$, so we conclude that $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}', \mathscr{C}_A')$. The opposite direction, verifying that $\Vdash_\Sigma \sigma' : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$ given $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}', \mathscr{C}_A')$, follows fairly easily:

$$\begin{aligned}
\sigma'(F[\overline{p_m^t}]\overline{p_n^x}) &= \sigma'(M)[\overline{p_m^t}]\overline{p_n^x} \\
&= \{\overline{p_m^t}/\overline{t_m}, \overline{p_n^x}/\overline{x_n}\}(\sigma'(G)[t_{i_1} \ldots t_{i_{m'}}]x_{k_1} \ldots x_{k_{n'}}) \\
&= \{p_{i_1}^t/t_{i_1}, \ldots, p_{i_m}^t/t_{i_m}, p_{k_1}^x/x_{k_1}, \ldots, p_{k_n}^x/x_{k_n}\}(\sigma'(G)[t_{i_1} \ldots t_{i_{m'}}]x_{k_1} \ldots x_{k_{n'}}) \\
&= \{\overline{q_m^t}/\overline{t_m}, \overline{q_n^x}/\overline{x_n}\}(\sigma'(G)[t_{i_1} \ldots t_{i_{m'}}]x_{k_1} \ldots x_{k_{n'}}) \\
&= \sigma'(N)[\overline{q_m^t}]\overline{q_n^x} \\
&= \sigma'(F[\overline{q_m^t}]\overline{q_n^x}).
\end{aligned}$$

Consider finally the case for **Flex-Flex-Diff** assume $\Vdash_\Sigma \sigma : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$. Then we have $\sigma(F_1) = \Lambda\overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda\overline{x_{n_1}} : \overline{A_{n_1}} \cdot M_1$ for some M_1 and $\sigma(F_2) = \Lambda\overline{t'_{m_2}} : \overline{K'_{m_2}} \cdot \lambda\overline{x'_{n_2}} : \overline{A'_{n_2}} \cdot M_2$ for some M_2, and furthermore $\{\overline{p_{m_1}^t}/\overline{t_{m_1}}, \overline{p_{n_1}^x}/\overline{x_{n_1}}\}M_1 = \sigma(F_1[\overline{p_{m_1}^t}]\overline{p_{n_1}^x}) = M = \sigma(F_2[\overline{q_{m_2}^t}]\overline{q_{n_2}^x}) = \{\overline{q_{m_2}^t}/\overline{t'_{m_2}}, \overline{q_{n_2}^x}/\overline{x'_{n_2}}\}M_2$ for some M. Let $(L''z)$ be a located occurrence of a term parameter in M (i.e. universally quantified in the local quantifier context). Since $z \notin FV(M_1)$, $z \notin FV(M_2)$ (by the scoping rules for the λ-calculus[5]), we must have $\{\overline{p_{m_1}^t}/\overline{t_{m_1}}, \overline{p_{n_1}^x}/\overline{x_{n_1}}\}(Lx_k) = \{\overline{q_{m_2}^t}/\overline{t'_{m_2}}, \overline{q_{n_2}^x}/\overline{x'_{n_2}}\}(L'x'_{k'})$ for some unique k, k', L, L'. So $Lp_k^x = (LL_k)z = L''z = (L'L'_{k'})z = L'q_{k'}^x$, giving us that $LL_k = L'L'_{k'}$. There are two cases to consider:

(i) If $L_k = L'_{k'}$ then $L = L'$. Since $p_k^x = q_{k'}^x$, we have $p_k^x \in P'$, $q_{k'} \in Q'$ and $q_{k'}^x \leqslant p_k$, $p_k \leqslant q_{k'}^x$. Then $\varphi_x(k'') = \langle k', k\rangle = \langle \mathbf{2}(\psi_x(k'')), \mathbf{1}(\psi_x(k''))\rangle$ for some $k'' \in [|P' \cup Q'|]$. So we have $\tilde{\varphi}_x(k'') = x_{\mathbf{1}(\varphi_x(k''))} = x_k = x_{\mathbf{2}(\psi_x(k''))}$ and $\tilde{\psi}_x(k'') = x'_{\mathbf{1}(\psi_x(k''))} = x'_{k'} = x'_{\mathbf{2}(\varphi_x(k''))}$.

(ii) If $L_k \neq L'_{k'}$ then $L_k < L'_{k'}$ or $L'_{k'} < L_k$. In the former case we have $p_k^x \notin P'$ and $q_{k'}^x \notin Q'$, so $L''_k L_k = L'_{k'}$ for some L''_k. So $LL_k = L'L'_{k'} = L'L''_k L_k$, giving us that $L = L'L''_k$. Also $\psi_x(k'') = \langle k, k'\rangle$ for some $k'' \in [|P' \cup Q'|]$, $k'' \notin dom(\varphi_x)$. Then $\tilde{\varphi}_x(k'') = L''_k x_{\mathbf{1}(\psi_x(k''))} = L''_k x_k$ and $\tilde{\psi}_x(k'') = x'_{\mathbf{2}(\psi_x(k''))} = x'_{k'}$.

In the latter case we have $L'_{k'} < L_k$. So $L''_{k'} L'_{k'} = L_k$ for some $L''_{k'}$. So $L'L'_{k'} = LL_k = LL''_{k'}L'_{k'}$, giving us that $L' = LL''_{k'}$. Also $\varphi_x(k'') = \langle k', k\rangle$ for some $k'' \in [|P' \cup Q'|]$, $k'' \notin dom(\psi_x)$. Then $\tilde{\varphi}_x(k'') = x_{\mathbf{2}(\varphi_x(k''))} = x_k$ and $\tilde{\psi}_x(k'') = L''_{k'} x_{\mathbf{1}(\varphi_x(k''))'} = L''_{k'} x'_{k'}$.

Recall the definitions of $\{\langle t''_k, K''_k\rangle\}_{k'' \in [m]} \subseteq \{\langle t_k, K_k\rangle\}_{k \in [m_1]} \cup \{\langle t'_k, K'_k\rangle\}_{k \in [m_2]}$ and $\{A''_k\}_{k'' \in [n]} \subseteq \{A_k\}_{k \in [n_1]} \cup \{A'_k\}_{k \in [n_2]}$ from Section 4.3. Now define M'_1 to be M_1 with occurrences of $(L''_k x_k)$ replaced by $x'_{k'}$ if $\langle k, k'\rangle$ is in the range of ψ_x, $p_k^x < q_{k'}^x$ and $L''_k p_k^x = q_{k'}^x$. Similarly define M'_2 to be M_2 with occurrences of $(L''_{k'} x'_{k'})$ replaced by x_k if

[5] So $z = \mathtt{source}(p_k^x)$ and $z = \mathtt{source}(q_{k'}^x)$ for some $k \in [n_1]$, $k' \in [n_2]$.

$\langle k', k\rangle$ is in the range of φ_x, $q^x_{k'} \leqslant p^x_k$ and $L''_{k'} q^x_{k'} = p^x_k$. We claim that $\{\tilde{\varphi}_x(1)/x''_1, \ldots, \tilde{\varphi}_x(n)/x''_n\} M'_1 = M_1$. If $k'' \in dom(\varphi_x)$, so $\varphi_x(k'') = \langle k', k\rangle$ for some k', k, then $x''_{k''} = x_{2(\psi(k''))} = x_k = \tilde{\varphi}_x(k'')$. If $k'' \notin dom(\varphi_x)$ then $\psi^x(k'') = \langle k, k'\rangle$ for some k, k' such that $q^x_{k'} = L''_k p^x_k$ For some L''_k. Then $x''_{k''} = x_{2(\psi(k''))} = x'_{k'} = L''_k x_k = L''_k x_{1(\psi(k''))} = \tilde{\varphi}_x(k'')$. Similarly we have that $\{\tilde{\psi}_x(1)/x''_1, \ldots, \tilde{\psi}_x(n)/x''_n\} M'_2 = M_2$. Now define $\sigma' = \sigma \cup \{(\Lambda \overline{t''_m} : \overline{K''_m} \cdot \lambda \overline{x''_n} : \overline{A''_n} \cdot M'_1)/G\}$. Then we have:

$$\begin{aligned}
\sigma'(F_1) &= \Lambda \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda \overline{x_{n_1}} : \overline{A_{n_1}} \cdot M_1 \\
&= \Lambda \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda \overline{x_{n_1}} : \overline{A_{n_1}} \cdot \{\tilde{\varphi}_x(1)/x''_1, \ldots, \tilde{\varphi}_x(n)/x''_n\} M'_1 \\
&= \Lambda \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda \overline{x_{n_1}} : \overline{A_{n_1}} \cdot \sigma'(G)[\tilde{\varphi}_t(1) \ldots \tilde{\varphi}_t(m)] \tilde{\varphi}_x(1) \ldots \tilde{\varphi}_x(n) \\
&= \sigma'(\Lambda \overline{t_{m_1}} : \overline{K_{m_1}} \cdot \lambda \overline{x_{n_1}} : \overline{A_{n_1}} \cdot G[\tilde{\varphi}_t(1) \ldots \tilde{\varphi}_t(m)] \tilde{\varphi}_x(1) \ldots \tilde{\varphi}_x(n))
\end{aligned}$$

where we use the fact that $\tilde{\varphi}_t(i) = t''_i$ (from the definition of the **Flex-Flex-Diff** rule). Similarly we have $\sigma'(F_2) = \sigma'(\Lambda \overline{t'_{m_2}} : \overline{K'_{m_2}} \cdot \lambda \overline{x'_{n_2}} : \overline{A'_{n_2}} \cdot G[\tilde{\psi}_t(1) \ldots \tilde{\psi}_t(m)] \tilde{\psi}_x(1) \ldots \tilde{\psi}_x(n))$, so we conclude that $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}', \mathscr{C}'_A)$.

For the other direction assume $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}', \mathscr{C}'_A)$ for some σ'. For each $k'' \in [n]$, where $n = |P' \cup \mathcal{Q}'|$, we have two cases to consider:

(i) If $k'' \in dom(\varphi_x)$, then $\varphi_x(k'') = \langle k', k\rangle$ for some k', k. Then $\tilde{\varphi}_x(k'') = x_k$, so $\{\overline{p^t_{m_1}/t_{m_1}}, \overline{p^x_{n_1}/x_{n_1}}\} \tilde{\varphi}_x(k'') = p^x_k$. On the other hand $\tilde{\psi}_x(k'') = L''_{k'} x'_{k'}$ for some $L''_{k'}$ such that $p^x_k = L''_{k'} q^x_{k'}$, so $\{\overline{q^t_{m_2}/t'_{m_2}}, \overline{q^x_{n_2}/x'_{n_2}}\} \tilde{\psi}_x(k'') = L''_{k'} q^x_{k'} = p^x_k$.

(ii) If $k'' \notin dom(\varphi_x)$, so $k'' \in dom(\psi_x)$ and $\psi_x(k'') = \langle k, k'\rangle$ for some k, k'. Then $\tilde{\varphi}_x(k'') = L''_k x_k$ for some L''_k such that $q^x_{k'} = L''_k p^x_k$, so $\{\overline{p^t_{m_1}/t_{m_1}}, \overline{p^x_{n_1}/x_{n_1}}\} \tilde{\varphi}_x(k'') = L''_k p^x_k = q^x_{k'}$. On the other hand $\tilde{\psi}_x(k'') = x'_{k'}$, so $\{\overline{q^t_{m_2}/t'_{m_2}}, \overline{q^x_{n_2}/x'_{n_2}}\} \tilde{\psi}_x(k'') = q^x_{k'}$.

Then

$$\begin{aligned}
\sigma'(F_1[\overline{p^t_{m_1}}]\overline{p^x_{n_1}}) &= \sigma'(G)[\{\overline{p^t_{m_1}/t_{m_1}}\} \tilde{\varphi}_t(1) \ldots \{\overline{p^t_{m_1}/t_{m_1}}\} \tilde{\varphi}_t(m)] \\
&\quad \cdot \{\overline{p^t_{m_1}/t_{m_1}}, \overline{p^x_{n_1}/x_{n_1}}\} \tilde{\varphi}_x(1) \ldots \{\overline{p^t_{m_1}/t_{m_1}}, \overline{p^x_{n_1}/x_{n_1}}\} \tilde{\varphi}_x(n) \\
&= \sigma'(G)[\{\overline{q^t_{m_2}/t'_{m_2}}\} \tilde{\varphi}_t(1) \ldots \{\overline{q^t_{m_2}/t'_{m_2}}\} \tilde{\varphi}_t(m)] \\
&\quad \cdot \{\overline{q^t_{m_2}/t'_{m_2}}, \overline{q^x_{n_2}/x'_{n_2}}\} \tilde{\varphi}_x(1) \ldots \{\overline{q^t_{m_2}/t'_{m_2}}, \overline{q^x_{n_2}/x'_{n_2}}\} \tilde{\varphi}_x(n) \\
&= \sigma'(F_2[\overline{q^t_{m_2}}]\overline{q^x_{n_2}}).
\end{aligned}$$

So we conclude that $\Vdash_\Sigma \sigma' : \mathcal{Q}(\mathscr{C}, \mathscr{C}_A)$. □

Lemma 27. *Given a valid initial configuration* $(\mathcal{Q}; \mathscr{C}; \top)$, *if* $(\mathcal{Q}; \mathscr{C}; \top) \Rightarrow (\mathcal{Q}'; \mathscr{C}'; \mathscr{C}'_A)$ *where* $\mathscr{C}'$ *is not empty, and no rules are applicable, then there is no substitution* σ *such that* $\Vdash_\Sigma \sigma : \mathcal{Q}(\mathscr{C})$.

Proof. We verify that if no rule is applicable for $(\mathcal{Q}'; \mathscr{C}'; \mathscr{C}'_A)$, then there is no satisfying substitution for the corresponding unification formula $\mathcal{Q}'(\mathscr{C}', \mathscr{C}'_A)$. The result then

follows as a corollary of the previous theorem. For the decomposition rules this is straightforward. For the **Rigid-Rigid** rule, type-checking does not guarantee that the argument lists have equal length; consider a constant $\mathsf{c} \in \Delta t : \mathsf{Type} \cdot t \to t$ and the constraint $(\mathsf{c}[\mathsf{nat}]3) = (\mathsf{c}[\mathsf{nat} \to \mathsf{nat}](\lambda x : \mathsf{nat} \cdot x)3)$. However, an induction on the length of two rigid terms verifies that they must have equal length if they are unifiable.

The occurs checks for the **Flex-Const** and **Flex-Param** rules are justified by an argument based on the size of equated terms. If σ is a unifying substitution for $F[\overline{p^t}]\overline{p^v} = M$ with M a rigid atom and $F \in FV(M)$, then we argue by induction on the number of outermost pair constructors and λ-abstractions in $\sigma(F)$. If this number is zero then we must have $|\sigma(F)| = |\sigma(M)|$ which is impossible since all terms are finite trees. If the number of pair constructors is zero or the length of the locator applied to the occurrence of F in M is zero, then the same argument applies. If the number of outermost pair constructors for $\sigma(F)$ is non-zero and the length of the locator applied to the occurrence of F in M is non-zero, then the locator is of the form Li. Let $\Lambda\bar{t} : \overline{K} \cdot \lambda\bar{x} : \overline{A} \cdot (M_1, M_2) = \sigma(F)$ and invoke the induction hypothesis on $\Lambda\bar{t} : \overline{K} \cdot \lambda\bar{x} : \overline{A} \cdot M_i$ and M. □

Definition 28 (*Answer substitution*). Given a final configuration of the unification algorithm $(\mathcal{Q}; \top; \mathscr{C}_A)$, with $\mathscr{C}_A = (t_i = A_i)_{i \in [m]}, (x_j = M_j)_{j \subset [n]}$, define the *answer substitution* to be

$$\theta = \{t_i \mapsto A_i \mid i \in [m]\} \cup \{u \mapsto u \mid u \in dom(\mathcal{Q}) - \{t_i\}_i\}$$
$$\cup \{x_j \mapsto M_j \mid j \in [n]\} \cup \{y \mapsto y \mid y \in dom(\mathcal{Q}) - \{x_j\}_j\}.$$

Theorem 29. Given a valid initial configuration $(\mathcal{Q}; \mathscr{C}; \top)$, the following are equivalent:

(i) $\Vdash_\Sigma \sigma : (\mathcal{Q}; \mathscr{C}; \top)$.

(ii) $(\mathcal{Q}; \mathscr{C}; \top) \stackrel{*}{\Rightarrow} (\mathcal{Q}'; \top, \mathscr{C}_A)$ with answer substitution θ, and for some substitution σ' such that $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}_A)$, $\sigma(x) = \sigma'(\theta(x))$ for all existentially quantified term variables x in $\mathcal{Q}$ (similarly for existentially quantified type variables t in $\mathcal{Q}$).

Proof. For completeness ("only if"), Theorem 17 verifies that the algorithm must terminate. If the final configuration is not in solved form, then Lemma 27 ensures that there is no unifying substitution for the original constraint list. Since σ is such a unifying substitution, the algorithm terminates with a configuration $(\mathcal{Q}'; \top; \mathscr{C}_A)$ in solved form. The proof of Theorem 26 constructs, by induction on the execution steps of the algorithm, a substitution $\sigma' \supseteq \sigma$ such that $\Vdash_\Sigma \sigma' : \mathcal{Q}'(\mathscr{C}'_A)$. Let θ be the answer substitution for the final configuration, then for all existentially quantified variables x in $\mathcal{Q}$ with an answer constraint $x = M$ in $\mathscr{C}_A$, $\theta(x) = M$; $\theta(x) = x$ otherwise. Then $\sigma(x) = \sigma'(x) = \sigma'(M) = \sigma'(\theta(x))$. Similarly for all existentially quantified type variables t.

For soundness ("if"), let $\sigma = \sigma'$, then Theorem 26 verifies that $\Vdash_\Sigma \sigma' : (\mathcal{Q}; \mathscr{C}; \top)$. □

7. Conclusions

We have presented an extension of Miller's algorithm for a decidable subset of higher-order unification [14]. This extension enriches the underlying λ-calculus with both impredicative polymorphism and product types. The extension enjoys the same desirable properties as Miller's restriction on unification problems for the simple typed λ-calculus, namely termination and most general unifiers. We extend Miller's original pattern restriction by allowing locators applied to the parameters in an application of a free function variable; the presence of locators allows variables to be repeated in the argument list, provided they are in the application scope of incomparable locators. This extension of Miller's original patterns is the basis of the definition of a higher-order form of the $\lambda\sigma$-calculus [1, 7, 19].

Our extension fundamentally alters the nature of the original unification problem identified by Miller. Because it relies on a very weak form of β-reduction and does not require type information in its execution, Miller's algorithm is in fact applicable to restricted unification in the untyped (or, universally typed) λ-calculus. On the other hand our algorithm relies fundamentally on well-typedness for its correctness. For example consider the (untyped) constraint:

$$\lambda x \cdot \mathsf{fst}(Fx) = \lambda x \cdot F(\mathsf{fst}\, x).$$

In fact there are an infinite number of incomparable unifying substitutions for this constraint, mapping $F \mapsto \lambda y \cdot \mathbf{1}^n(y)$ for any $n \in \omega$. This illustrates that any attempt to use the extended pattern restriction in a system with recursive types must be made with care.

Even adapting the algorithm to a calculus with cumulative predicative polymorphism introduces non-trivial complications. For example in a calculus such as Luo's Extended Calculus of Constructions [13], with cumulative type universes $\mathsf{Type}_0 \subseteq \mathsf{Type}_1 \subseteq \cdots \subseteq \mathsf{Type}_i \subseteq \mathsf{Type}_{i+1} \subseteq \cdots$ and dependent product types $(M,N) \in \Sigma t : A \cdot B$, the following constraint is well-typed (both sides are of type $\Sigma t : \mathsf{Type}_1 \cdot t$) yet admits a non-rigid recursive self-reference of F:

$$\exists F : (\Sigma t : \mathsf{Type}_0 \cdot t) \cdot F = (\Sigma t : \mathsf{Type}_0 \cdot t, F).$$

Allowing such circularities would invalidate the termination argument in Section 5, and would require considerably more complicated reasoning.

A major issue for the usefulness of the algorithm is undoubtedly the provision of an efficient implementation. Qian has provided an algorithm for Miller's original algorithm which is linear in time and space [28]. The Elf language employs an efficient, although not purely functional, implementation of Miller's algorithm [24], while Nipkow has provided a functional implementation in ML [21]. It remains to be seen if these implementations may be adapted easily to extended patterns, in particular with locators applied to parameters in a flexible application. The most complicated rules for the algorithm are the flex-flex rules, particularly the **Flex-Flex-Diff** rule. An efficient compilation technique for matching for extended patterns, based on renaming

$$\times F, \times I \quad \frac{\Gamma \rhd_\Sigma A \in \mathsf{Type} \quad \Gamma \rhd_\Sigma B \in \mathsf{Type}}{\Gamma \rhd_\Sigma A \times B \in \mathsf{Type}} \quad \frac{\Gamma \rhd_\Sigma M \in A \quad \Gamma \rhd_\Sigma N \in B}{\Gamma \rhd_\Sigma (M,N) \in A \times B}$$

$$\times E, \times \eta \quad \frac{\Gamma \rhd_\Sigma M \in A_1 \times A_2}{\Gamma \rhd_\Sigma \mathsf{fst}\, M \in A_1} \quad \frac{\Gamma \rhd_\Sigma M \in A_1 \times A_2}{\Gamma \rhd_\Sigma \mathsf{snd}\, M \in A_2} \quad \frac{\Gamma \rhd_\Sigma M \in A \times B}{\Gamma \rhd_\Sigma (\mathsf{fst}\, M, \mathsf{snd}\, M) = M}$$

$$\times \beta \quad \frac{\Gamma \rhd_\Sigma M_1 \in A_1 \quad \Gamma \rhd_\Sigma M_2 \in A_2}{\Gamma \rhd_\Sigma \mathsf{fst}\,(M_1,M_2) = M_1} \quad \frac{\Gamma \rhd_\Sigma M_1 \in A_1 \quad \Gamma \rhd_\Sigma M_2 \in A_2}{\Gamma \rhd_\Sigma \mathsf{snd}\,(M_1,M_2) = M_2}$$

Fig. 7. Type rules for products with projectors.

substitutions and treating the **Flex-Const** and **Flex-Param** as default rules, is presented in [2]. If extended patterns are used as the basis for higher-order rewrite systems with explicit substitutions, the full unification algorithm remains useful as a basis for checking higher-order criticial pairs in such a system.

Appendix

A. The projector calculus

In this appendix we give a formulation of product types, for the System $\mathbf{F}^\omega$ calculus presented in Section 2, which may be more familiar to the reader than the locator calculus presented in that section. We term this the *projector calculus*. We then give a translation from the locator calculus into the projector calculus, verifying the equivalence of the two systems.

The type rules for the projector calculus are given in Fig. 7. The following lemma considers the soundness of the locator calculus typing and equality rules using a transformation into the projector calculus.

Lemma 30 (Soundness of term locator rules). *The translation from the term locator calculus to the term projector calculus is given in Fig.* 8 *by induction on typing derivations. The typing rules and conversion rules for locators are sound with respect to the original conversion rules with this translation, in the following sense*:
(i) *If* $\Gamma \vdash_\Sigma M \in A$ *then* $\Gamma \vdash_\Sigma [\![\Gamma \vdash_\Sigma M \in A]\!] \in A$.
(ii) *If* $\Gamma \vdash_\Sigma M \in A$ *and* $\Gamma \vdash_\Sigma M = N$, *then* $\Gamma \vdash_\Sigma [\![\Gamma \vdash_\Sigma M \in A]\!] = [\![\Gamma \vdash_\Sigma N \in A]\!]$.

Proof. We verify the rules for permuting locator applications with Λ-abstractions and polymorphic applications, by induction on the length of the locator. For Λ-abstractions we have:

$$(L\,\mathbf{1})\,(\Lambda t : K \cdot M) = (L\,(\lambda f : (\Delta t : K \cdot A) \cdot \Lambda t : K \cdot A\overline{t_m} : \overline{K_m} \cdot \lambda \overline{x_n} : \overline{A_n} \cdot \mathsf{fst}\,(f[t][\overline{B_m}]\overline{x_n})))(\Lambda t : K \cdot M)$$

$$
\begin{aligned}
[\![\mathbf{1}]\!]_{\overline{K_m},\overline{A_n}}(A;\overline{B_m}) &\stackrel{\text{def}}{=} \lambda f : A \cdot \Lambda \overline{t_m} : \overline{K_m} \cdot \lambda \overline{x_n} : \overline{A_n} \cdot \mathsf{fst}\,(f[\overline{B_m}]\overline{x_n}) \\
[\![\mathbf{2}]\!]_{\overline{K_m},\overline{A_n}}(A;\overline{B_m}) &\stackrel{\text{def}}{=} \lambda f : A \cdot \Lambda \overline{t_m} : \overline{K_m} \cdot \lambda \overline{x_n} : \overline{A_n} \cdot \mathsf{snd}\,(f[\overline{B_m}]\overline{x_n}) \\
[\![\Gamma \vdash_\Sigma (iL)\, M[\overline{B_m}]\overline{N_n} \in B]\!] &\stackrel{\text{def}}{=} ([\![i]\!]_{\overline{K_m},\overline{A_n}}(B';\overline{B_m})) \\
&\qquad [\![\Gamma \vdash_\Sigma L\, M[\overline{B_n}]\overline{N_n} \in B']\!] \\
[\![\Gamma \vdash_\Sigma (M,N) \in A \times B]\!] &\stackrel{\text{def}}{=} ([\![\Gamma \vdash_\Sigma M \in A]\!],[\![\Gamma \vdash_\Sigma N \in B]\!]) \\
[\![\Gamma \vdash_\Sigma M[B] \in B']\!] &\stackrel{\text{def}}{=} [\![\Gamma \vdash_\Sigma M \in \Delta t : K \cdot B'']\!][\![\Gamma \vdash_\Sigma B \in K]\!] \\
[\![\Gamma \vdash_\Sigma \Lambda t : K \cdot M \in \Delta t : K \cdot B]\!] &\stackrel{\text{def}}{=} \Lambda t : K \cdot [\![\Gamma, t : K \vdash_\Sigma M \in B]\!] \\
[\![\Gamma \vdash_\Sigma M\, N \in B]\!] &\stackrel{\text{def}}{=} [\![\Gamma \vdash_\Sigma M \in A \to B]\!]([\![\Gamma \vdash_\Sigma N \in A]\!]) \\
[\![\Gamma \vdash_\Sigma \lambda x : A \cdot M \in A \to B]\!] &\stackrel{\text{def}}{=} \lambda x : A \cdot [\![\Gamma, x : A \vdash_\Sigma M \in B]\!] \\
[\![\Gamma \vdash_\Sigma \mathbf{id}\, M \in A]\!] &\stackrel{\text{def}}{=} [\![\Gamma \vdash_\Sigma M \in A]\!] \\
[\![\Gamma \vdash_\Sigma x \in A]\!] &\stackrel{\text{def}}{=} x \\
[\![\Gamma \vdash_\Sigma \mathsf{c} \in A]\!] &\stackrel{\text{def}}{=} \mathsf{c}
\end{aligned}
$$

Fig. 8. Translation from the locator calculus to the projector calculus.

$$
\begin{aligned}
&= L\,(\Lambda t : K \cdot \Lambda \overline{t_m} : \overline{K_m} \cdot \lambda \overline{x_n} : \overline{A_n} \cdot \mathsf{fst}\,((\Lambda t : K \cdot M)[t][\overline{B_m}]\overline{x_n})) \\
&= L\,(\Lambda t : K \cdot \Lambda \overline{t_m} : \overline{K_m} \cdot \lambda \overline{x_n} : \overline{A_n} \cdot \mathsf{fst}\,(M[\overline{B_m}]\overline{x_n})) \\
&= L\,(\Lambda t : K \cdot (\lambda f : A \cdot \Lambda \overline{t_m} : \overline{K_m} \cdot \lambda \overline{x_n} : \overline{A_n} \cdot \mathsf{fst}\,(f[\overline{B_m}]\overline{x_n}))\, M) \\
&= L\,(\Lambda t : K \cdot \mathbf{1}\, M) \\
&= \Lambda t : K \cdot L\,(\mathbf{1}\, M) \quad \text{using the induction hypothesis} \\
&= \Lambda t : K \cdot (L\,\mathbf{1})\, M.
\end{aligned}
$$

For applications we have:

$$
\begin{aligned}
(L\,\mathbf{1})\,(M\,[A]) &= (L\,(\lambda f : A \cdot \Lambda \overline{t_m} : \overline{K_m} \cdot \lambda \overline{x_n} : \overline{A_n} \cdot \mathsf{fst}\,(f[\overline{B_m}]\overline{x_n})))(M\,[A]) \\
&= L\,(\Lambda \overline{t_m} : \overline{K_m} \cdot \lambda \overline{x_n} : \overline{A_n} \cdot \mathsf{fst}\,(M\,[A][\overline{B_m}]\overline{x_n})) \\
&= L\,((\lambda y : B \cdot \Lambda \overline{t_m} : \overline{K_m} \cdot \lambda \overline{x_n} : \overline{A_n} \cdot \mathsf{fst}\,(My[\overline{t_m}]\overline{x_n}))N) \\
&= L\,((\lambda f : (\Delta t : K \cdot A') \cdot \Lambda t : K \cdot \Lambda \overline{t_m} : \overline{K_m} \cdot \lambda \overline{x_n} : \overline{A_n} \cdot \\
&\qquad \mathsf{fst}\,(f\,[A]\,[\overline{B_m}]\overline{x_n}))\, M\,[A]) \\
&= L\,((\mathbf{1}\, M)\,[A]) \\
&= (L\,(\mathbf{1}\, M))\,[A] \quad \text{using the induction hypothesis} \\
&= ((L\,\mathbf{1})\, M)\,[A]. \qquad \square
\end{aligned}
$$

Corollary 31. *The following rules are admissible*:

$$\times\beta \quad \frac{\Gamma \rhd_\Sigma (L\,M_1) \in \Delta\overline{t_m} : \overline{K_m} \cdot \overline{A_n} \to B_1 \quad \Gamma \rhd_\Sigma (L\,M_2) \in \Delta\overline{t_m} : \overline{K_m} \cdot \overline{A_n} \to B_2}{\Gamma \rhd_\Sigma (Li)(\Lambda\overline{t_m} : \overline{K_m} \cdot \lambda\overline{x_n} : \overline{A_n} \cdot (M_1[\overline{t_m}]\overline{x_n}, M_2[\overline{t_m}]\overline{x_n})) = (L\,M_i)}$$

$$\times\eta \quad \frac{\Gamma \rhd_\Sigma L\,M \in \Delta\overline{t_m} : \overline{K_m} \cdot \overline{A_n} \to B_1 \times B_2}{\Gamma \rhd_\Sigma \Lambda\overline{t_m} : \overline{K_m} \cdot \lambda\overline{x_n} : \overline{A_n} \cdot ((\mathbf{1}\,L)\,M[\overline{t_m}]\overline{x_n}, (\mathbf{2}\,L)\,M[\overline{t_m}]\overline{x_n}) = L\,M}.$$

The translation in the reverse direction, from the projector calculus to the locator calculus, is obvious, translating projectors $\mathrm{fst}\,M$ and $\mathrm{snd}\,M$ to selectors $\mathbf{1}\,M$ and $\mathbf{2}M$, respectively. Combining this with the previous results gives the following form of completeness for the locator calculus:

Theorem 32. (i) $\vdash \Gamma\mathbf{env}_\Sigma$ *if and only if* $\vdash \Gamma\mathbf{env}_\Sigma$.

(ii) $\Gamma \vdash_\Sigma M \in A$ *if and only if* $\Gamma \vdash_\Sigma [\![\Gamma \vdash_\Sigma M \in A]\!] \in A$.

(iii) $\Gamma \vdash_\Sigma M = N$ *if and only if* $\Gamma \vdash_\Sigma [\![\Gamma \vdash_\Sigma M \in A]\!] = [\![\Gamma \vdash_\Sigma N \in A]\!]$.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, J.-J. Lévy, Explicit substitutions, J. Funct. Programming 1 (4) (1991) 375–416.

[2] D. Duggan, Higher-order substitutions, Information and Computation, to appear.

[3] D. Duggan, Metaprogramming with higher-order abstract syntax, products and polymorphism, submitted for publication.

[4] C. Elliott, Some extensions and applications of higher order unification, Technical Report ERGO-88-061, Carnegie-Mellon University, 1988; Ph.D. Thesis Proposal.

[5] C. Elliott, Higer-order unification with dependent types, Rewriting Techniques and Applications, Lecture Notes in Computer Science, vol. 355, Springer, Berlin, 1989.

[6] C.M. Elliott, Extensions and Applications of Higher-Order Unification, Ph.D. Thesis, Carnegie Mellon University, 1990; Available as Technical Report CMU-CS-90-134.

[7] J. Field, On laziness and optimality in lambda interpreters: tools for specification and analysis, in: Proc. ACM Symp. on Principles of Programming Languages, 1990, pp. 1–15.

[8] J. Gallier, On Girard's 'candidats de reducibilité', Technical report, University of Pennsylvania, Philadelphia, PA, 1990; appeared as a chapter in: P. Odifreddi (Ed.), Logic, and Comput. Science, Academic Press, New York, 1990.

[9] H. Geuvers, The Church–Rosser property for $\beta\eta$-reduction in typed λ-calculi, in: Proc. IEEE Symp. on Logic in Computer Science, Santa Cruz, California, IEEE Press, New York, 1992, pp. 453–460.

[10] J.-Y. Girard, Interprétation Fonctionelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur, Ph.D. Thesis, Thèse de Doctorat d'État, Paris, 1972.

[11] R. Harper, J. Mitchell, E. Moggi, Higher-order modules and the phase distinction, in: Proc. ACM Symp. on Principles of Programming Languages, Association for Computing Machinery, 1990, pp. 341–354.

[12] G. Huet, A unification algorithm for typed λ-calculus, Theoret. Comput. Sci. 1 (1975) 27–57.

[13] Z. Luo, ECC, an extended calculus of constructions, in: Proc. IEEE Symp. on Logic in Computer Science, IEEE Press, New York, 1989, pp. 385–395.

[14] D. Miller, A logic programming language with lambda-abstraction, function variables and simple unification, J. Logic Comput. 1 (4) (1991) 497–536.

[15] D. Miller, Unification of simply typed λ-terms as logic programming, in: Proc. Internat. Conf. on Logic Programming, MIT Press, Cambridge, MA, 1991, pp. 255–269.

[16] D. Miller, Unification under a mixed prefix, J. Symbolic Comput. 14 (1992) 321–358.

[17] G. Nadathur, D. Miller, An overview of λ-Prolog, in: Proc. Internat. Conf. on Logic Programming, MIT Press, Cambridge, MA, 1988, pp. 810–827.

[18] G. Nadathur, D. Miller, Higher-order Horn clauses, J. Assoc. Comput. Mach. 37 (4) (1990) 777–814.
[19] G. Nadathur, D.S. Wilson, A representation of lambda terms suitable for operations on their intensions, in: Proc. ACM Symp. on Lisp and Functional Progamming, Association for Computing Machinery, New York, 1990, pp. 341–348.
[20] T. Nipkow, Higher order critical pairs, in: Proc. IEEE Symp. on Logic in Computer Science, IEEE Press, New York, 1991, pp. 342–349.
[21] T. Nipkow, Functional unification of higher-order patterns, in: Proc. IEEE Symp. on Logic in Computer Science, IEEE Press, New York, 1993.
[22] L. Paulson, Isabelle: the next 700 theorem provers, in: P. Odifreddi (Ed.), Logic and Computer Science, Academic Press, New York, 1990, pp. 361–385.
[23] F. Pfenning, Elf: a language for logic definition and verified metaprogramming, in: Proc. IEEE Symp. on Logic in Computer Science, IEEE Press, New York, 1989, pp. 313–322.
[24] F. Pfenning, Logic programming in the LF logical framework, in: G. Huet, G. Plotkin (Eds.), Logical Frameworks, Cambridge University Press, Cambridge, 1990, pp. 149–181.
[25] F. Pfenning, Unification and anti-unification in the calculus of constructions, in: Proc. IEEE Symp. on Logic in Computer Science, 1991.
[26] F. Pfenning, C. Elliott, Higher-order abstract syntax, in: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, Association for Computing Machinery, New York, 1988, pp. 199–208.
[27] D. Pym, Proofs, search and computation in general logic, Ph.D. Thesis, University of Edinburgh, Scotland, 1990; Available as CST-69-90, also published as ECS-LFCS-90-125.
[28] Z. Qian, Linear unification of higher-order patterns, in: M.-C. Gaudel, J.-P. Jouannaud (Eds.), Proc. Coll. on Trees in Algebra and Programming, Orsay, France, Lecture Notes in Computer Science, vol. 668, Springer, Berlin, 1993, pp. 391–405.